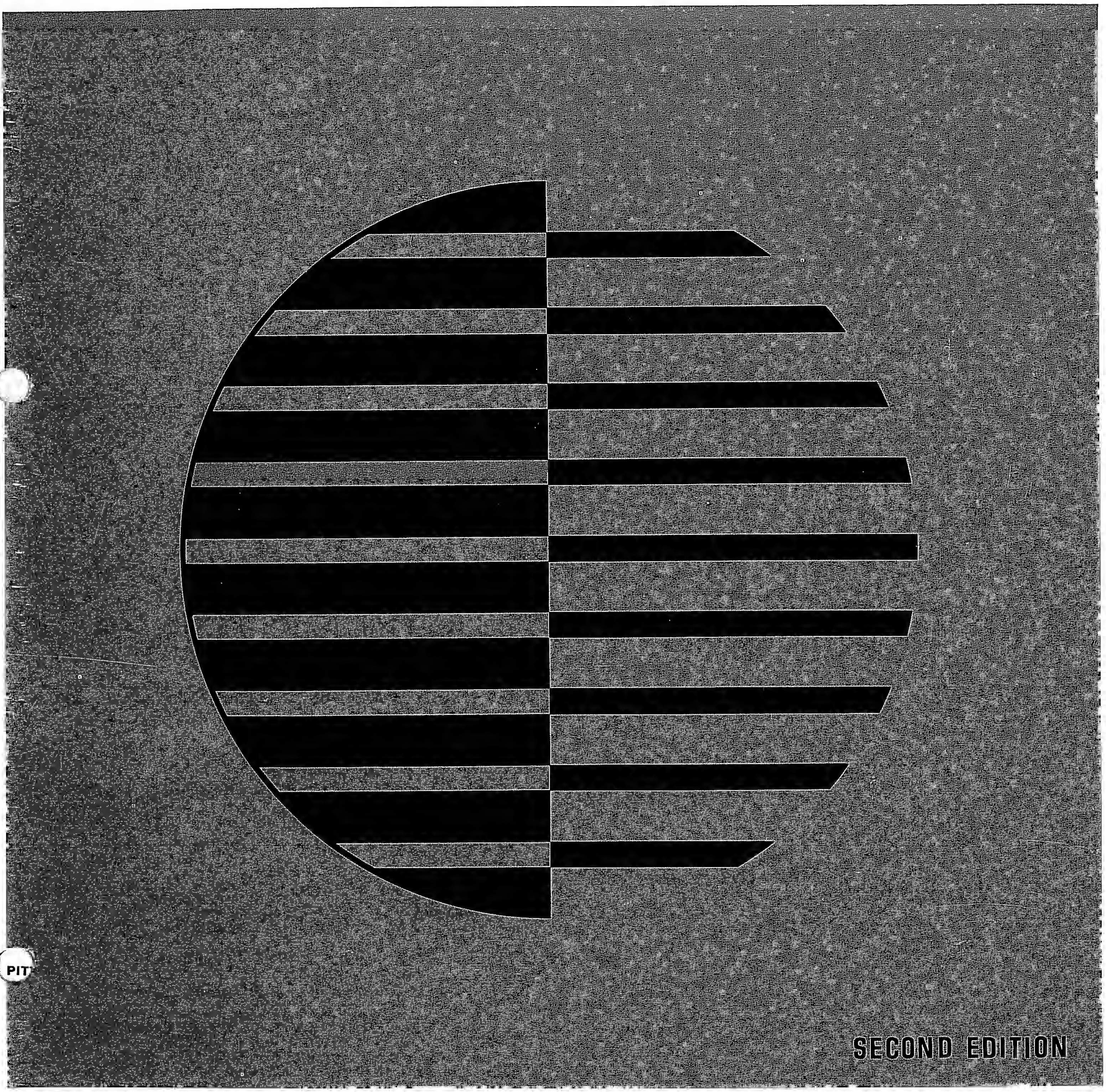
CONTROL DATA® 6600 Computer System Programming System/Reference Manual

Volume 3 FORTRAN 66



SECOND EDITION

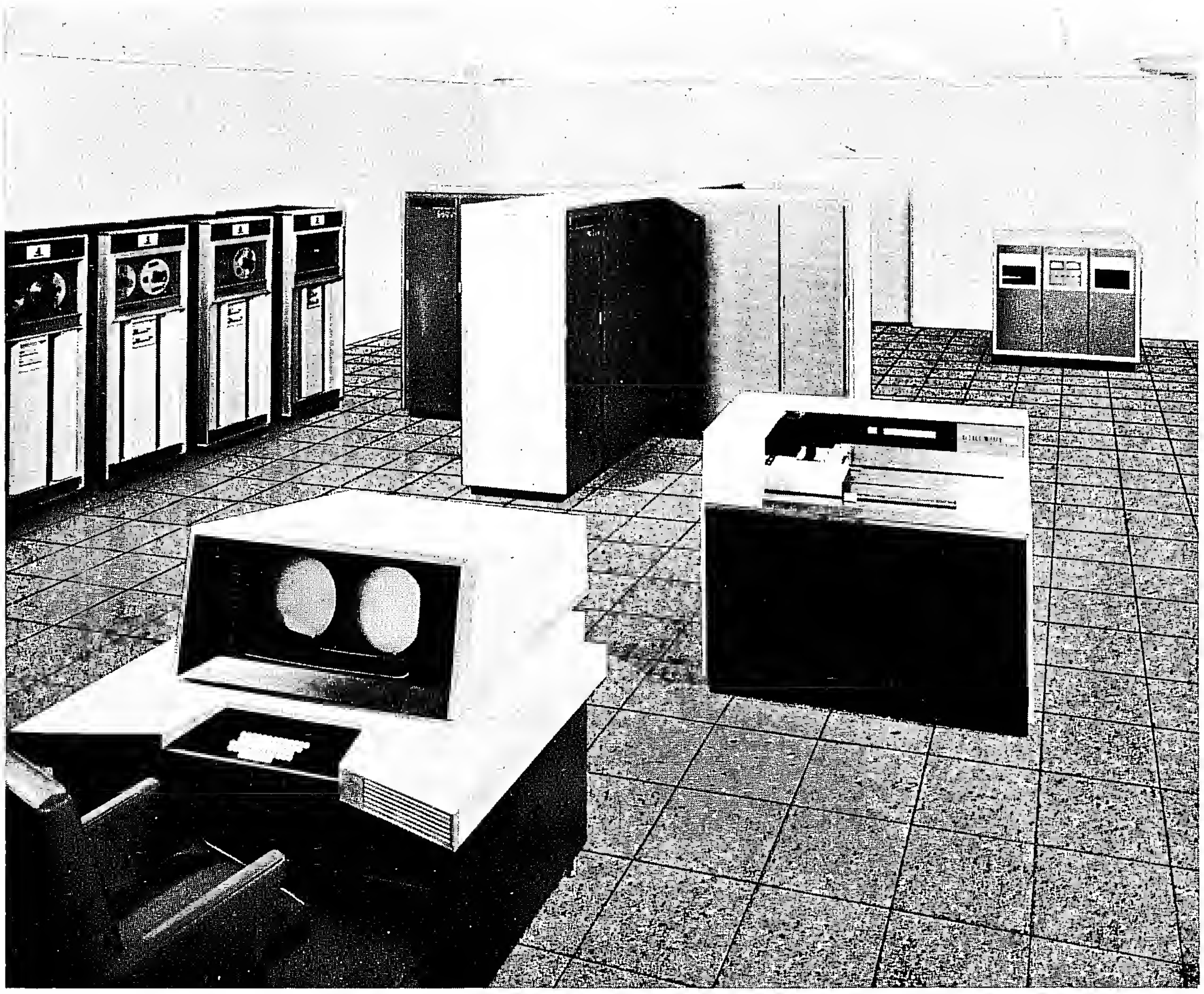
PREFACE

The FORTRAN*-66 language contains most of the features of FORTRAN-63. The FORTRAN-66 compiler adapts current techniques to the particular capabilities of the Control Data® 6600 Computer System. Emphasis has been placed on producing highly efficient object programs while maintaining the efficiency of compilation.

This reference manual was written as a text for advanced FORTRAN-66 classes and as a reference manual for programmers using the FORTRAN-66 system. The manual assumes a basic knowledge of the FORTRAN language.

*FORTRAN is an abbreviation for FORmula TRANslation and was originally developed for International Business Machine equipment.

TABLE OF CONTENTS



6600 COMPUTING SYSTEM

Main frame *(center)—contains 10 peripheral and control processors, central processor, central memory, some I/O synchronizers.*

Display console *(foreground)—includes a keyboard for manual input and operator control, and two 10-inch display tubes for display of problem status and operator directives.*

CONTROL DATA 607 tapes *(left front)—½ inch magnetic tape units for supplementary storage; binary or BCD data handled at 200, 556, or 800 bpi.*

CONTROL DATA 626 tapes *(left rear)—1-inch magnetic tape units for supplementary storage; binary data handled at 800 bpi.*

Disc file *(right rear)—Supplementary mass storage device holds 500 million bits of information.*

CONTROL DATA 405 card reader *(right front)—reads binary or BCD cards at 1200 card per minute rate.*

SYSTEM ORGANIZATION

The CONTROL DATA® 6600 is a large-scale, solid-state, general-purpose digital computing system. The advanced design techniques incorporated in the system provide for extremely fast solutions to data processing, scientific and control center problems.

Within the 6600 are eleven independent computers (Fig. 1). Ten of these are constructed with the peripheral and operating system in mind. These ten have separate memory and can execute programs independently of each other or the central processor. The eleventh computer, the central processor, is a very high-speed arithmetic device. The common element between these computers is the large central memory.

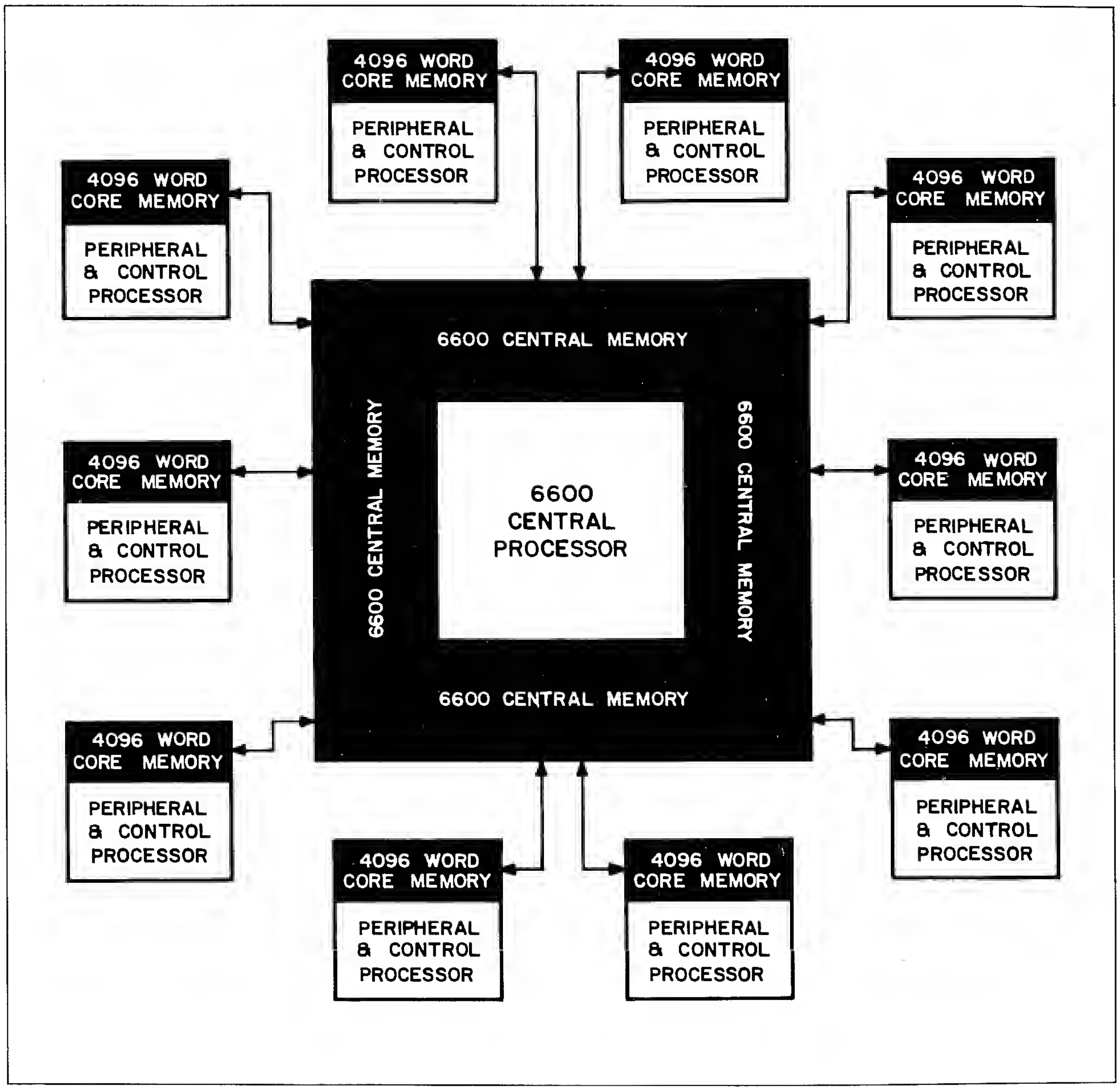
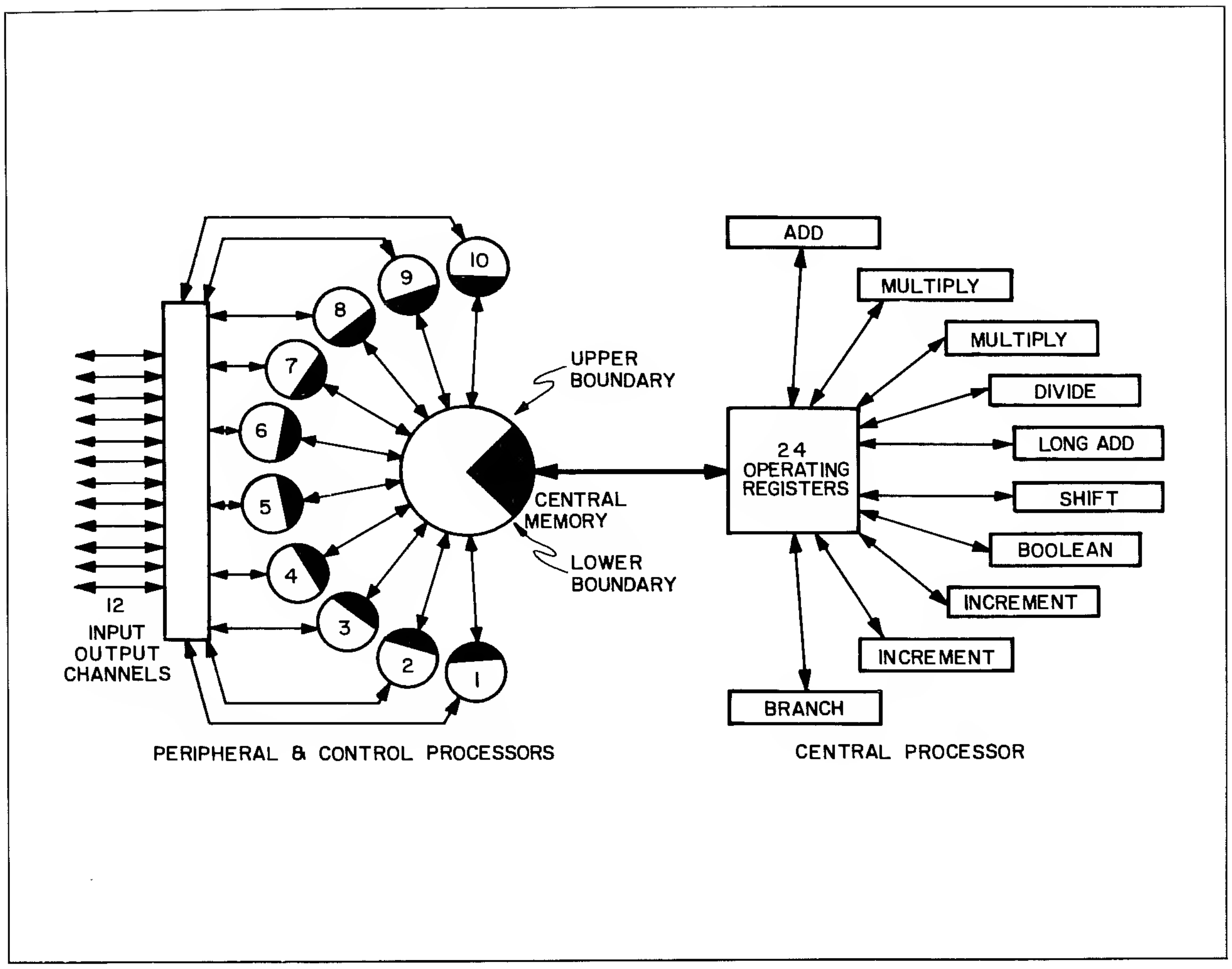


Figure 1 CONTROL DATA 6600



CENTRAL MEMORY

— 131,072 words

— 60-bit words

— Memory organized in 32 logically independent banks of 4096 words with corresponding multi-phasing of banks

— Random access, coincident-current, magnetic core

— One major cycle for read-write

— Maximum memory reference rate to all banks — one address/minor cycle

— Maximum rate of data flow to/from memory — one word/minor cycle

DISPLAY CONSOLE

— Two display tubes

— Modes
Character
Dot

— Character size
Large — 16 characters/line
Medium — 32 characters/line
Small — 64 characters/line

— Characters
26 alphabetic
10 numeric
11 special

Figure 2 BLOCK DIAGRAM OF 6600

1. ELEMENTS OF FORTRAN-66

1.1 CONSTANTS

Four basic types of constants are used in FORTRAN-66: integer, octal, floating point and Hollerith. Complex and double precision constants can be formed from floating point constants. The type of a constant is determined by its form. The computer word structure for each type is given in Appendix E.

1.1.1 INTEGER

Integer constants may consist of up to 18 decimal digits, in the range from $-(2^{59}-1)$ to $(2^{59}-1)$. Integer multiply and divide are executed in floating point modules. No double precision is performed on integer values; if multiply and divide are specified, operands and result will be taken modular 2^{47}.

Examples:

63	3647631
−247	2
314159265	464646464

1.1.2 OCTAL

Octal constants may consist of up to 20 octal digits. The form is:

n_1 --- n_iB

Examples:

```
77777777000000000000B
        7777700077777B
23232323232323232323B
                  77B
    777777777777777700B
```

1.1.3 FLOATING POINT

Real constants are represented by a string of up to 15 digits. A real constant may be expressed with a decimal point or with a fraction and an exponent representing a power of ten. The forms of real constants are:

nE n.n n. .n nE±s n.nE±s n.E±s .nE±s

n is the coefficient; s is the exponent to the base 10. The plus sign may be omitted for positive s. The range of s is 0 through 308.

Examples:

3.1415768	31.41592E−01
314.	.31415E01
.0749162	.31415E+01
314159E−05	

Double precision constants are represented by a string of up to 29 digits. The forms are:

nD±s .nD±s nD n.nD n.D .nD
n.nD±s n.D±s

n is the coefficient; s is the exponent to the base 10.

The D must always appear. The plus sign may be omitted for positive s; the range of s is 0 through 308.

Examples:

3.14159265358979323846 26D
3.1415D
3.1415D0
3141.598D−03
31415.D−04
379867524430111D+01

Complex constants are represented by pairs of real constants separated by a comma and enclosed in parentheses (R_1, R_2). The real part of a complex number is represented by R_1, the imaginary part by R_2. Either constant may be preceded by a − sign.

Examples:

FORTRAN-66 Representation	*Complex Number*
(1., 6.55)	1.+6.55i
(15., 16.7)	15.+16.7i
(−14.09, 1.654E−04)	−14.09+.0001654i
(0., −1.)	−i

1.1.4 HOLLERITH

A Hollerith constant is a string of alphanumeric characters of the form hHf; h is an unsigned decimal integer representing the length of the field f and must be 10 or less characters except when used in a format statement. Spaces are significant in the field f. When h is not a multiple of 10, the last computer word is left-justified with BCD spaces filling the remainder of the word.

An alternate form of a Hollerith constant is hRf. When h is less than or equal to 10, the computer word is right-justified with zero fill. When h is greater than 10 only the first 10 characters are retained and the excess characters are discarded.

Examples:

6HCOGITO	8RCDC 6600
4HERGO	8R **
3HSUM	1R)

1.2 VARIABLES

FORTRAN-66 recognizes simple and subscripted variables. A simple variable represents a single quantity; a subscripted variable represents a single quantity within an array of quantities. The variable is either defined in a TYPE declaration or determined by the first letter of the variable name. A first letter of I, J, K, L, M, or N indicates a fixed point (integer) variable; any other first letter indicates a floating point variable. However, a TYPE declaration overrides any other naming convention. (See Section 4-1.)

1.2.1 SIMPLE

A simple variable is the name of a storage area in which values can be stored. The variable is referenced by the location name; the value specified by the name is always the current value stored in that location.

Simple integer variables are identified by at least one or up to 8 alphabetic or numeric characters; the first must be I, J, K, L, M, or N. Any integer value in the range from $-(2^{59}-1)$ to $2^{59}-1$ may be assigned to a simple integer variable.

Examples:

N	NOODGE
K2SO4	M58
LOX	M 58

Since spaces are ignored in variable names, M58 and M 58 are identical.

Simple floating point variables are identified by at least one or up to 8 alphabetic or numeric characters; the first must be alphabetic and *not* I, J, K, L, M, or N. Any value from 10^{-308} to 10^{308} and zero can be assigned to a simple floating point variable. A TYPE declaration overrides any other naming convention.

Examples:

VECTOR	A65302
BAGELS	BATMAN

1.2.2 SUBSCRIPTED VARIABLE ARRAYS

A subscripted variable represents an element of an array. An array is a block of successive memory locations which is divided into areas for storage of variables. Each element of the array is referenced by the array name plus a subscript. The type of an array is determined by the array name or TYPE declaration. Arrays may have one, two, or three dimensions and the maximum number of elements is the product of the dimensions. A subscript can be an integer constant, an integer variable, or any integer expression. Any other constant, variable, or expression will be reduced to an integer value. The array name and its dimensions must be declared at the beginning of the program in a DIMENSION statement (section 4.2).

Array structure. Elements of arrays are stored by columns in ascending order of storage location. In the array declared as A(3,3,3):

$$\boxed{\begin{array}{ccc} A_{111} & A_{121} & A_{131} \\ A_{211} & A_{221} & A_{231} \\ A_{311} & A_{321} & A_{331} \end{array}}$$

$$\boxed{\begin{array}{ccc} A_{112} & A_{122} & A_{132} \\ A_{212} & A_{222} & A_{232} \\ A_{312} & A_{322} & A_{332} \end{array}}$$

$$\boxed{\begin{array}{ccc} A_{113} & A_{123} & A_{133} \\ A_{213} & A_{223} & A_{233} \\ A_{313} & A_{323} & A_{333} \end{array}}$$

The planes are stored in order, starting with the first, as follows:

$A_{111} \rightarrow L$	$A_{121} \rightarrow L+3$ ...	$A_{133} \rightarrow L+24$
$A_{211} \rightarrow L+1$	$A_{221} \rightarrow L+4$ ...	$A_{233} \rightarrow L+25$
$A_{311} \rightarrow L+2$	$A_{321} \rightarrow L+5$ ...	$A_{333} \rightarrow L+26$

Program errors may result if subscripts are larger than the dimensions initially declared for the array. A single subscript notation may be used for a two or three dimensional array if it does not exceed the product of the declared dimensions.

1.2.3 SUBSCRIPT FORMS

Subscripted integer variables, the elements of an integer array, may be assigned the same values as simple integer values. An integer array is named by an integer variable name (1 to 8 alphabetic or numeric characters, the first of which is I, J, K, L, M, or N), or is designated integer by a TYPE INTEGER statement.

```
NEURON (6, 8, 6)      L6034 (J, 3)
MORPH (20, 20)        N3 (1)
```

Subscripted floating point variables, the elements of a floating point array, may be assigned the same values as simple floating point variables. A floating point variable array is named with a floating point variable name (1 to 8 alphabetic or numeric characters, the first of which is alphabetic and not I, J, K, L, M, or N), or is designated floating point by a TYPE REAL statement.

```
TMESIS (6, 4, 7)      YCLEPT (46)
PST (6, 8)            SVELTE (37, 24, 35)
```

Any computable arithmetic expression may be used as a subscript.

```
A (I, J)                  A(MAXF(I, J, M) )
B (I+2, J+3, 2*K+1)       B(J, ATANF(J) )
Q (14)                    C(I+K)
P (KLIM, JLIM+5)
MOTZO  (3*K*ILIM+3.5)
SAM (J-6)                 WOW(I(J(K) ) )
```

The location of an array element with respect to the first element is a function of the maximum array dimensions and the type of the array. Given DIMENSION A(L, M, N) the location of A (i, j, k), with respect to the first element A of the array, is given by

$$A+\{i-1+L(j-1+M(k-1))\}*E$$

The quantity in braces is the subscript expression. If it is not an integer value, it is truncated after evaluation.

E is the element length, that is, the number of storage words required for each element of the array; for real and integer arrays, E=1. (See Appendix E)

1. Referring to the matrix in 1.2.2, the location of A (2,2,3) with respect to A (1,1,1) is:

$$\begin{aligned} \text{Locn}\,\{A(2,2,3)\} &= \text{Locn}\,\{A(1,1,1)\} \\ &\quad +\{2-1+3(1+3(2))\} \\ &= L+22 \end{aligned}$$

2. Given DIMENSION Z (5,5,5) and I=1, K=2, X=45°, A=7.29, B=1.62. The location, z, of Z (I*K, TANF(X), A−B) with respect to Z (1,1,1) is:

$$\begin{aligned} z &= \text{Locn}\,(Z(1,1,1)) + (2-1+5 \\ &\qquad ([\text{TANF}(X)] - 1 + 5[4.67])) \\ &= \text{Locn}\,(Z(1,1,1)) + (2-1+5 \\ &\qquad (1-1+5*4)) \\ &= \text{Locn}\,(Z(1,1,1)) + 101 \end{aligned}$$

FORTRAN-66 permits the following relaxation of the representation of subscripted variables:

Given $A(D_1,D_2,D_3)$:

where the D_i are integer constants.

then	A(I,J,K)	implies A(I,J,K)
	A(I,J)	implies A(I,J,1)
	A(I)	implies A(I,1,1)
	A	implies A(1,1,1)

similarly, for $A(D_1,D_2)$

A(I,J)	implies A(I,J)
A(I)	implies A(I,1)
A	implies A(1,1)

and for $A(D_1)$

A	implies A(1)

However, the elements of a single-dimension array

A(D_1) may not be referred to as A(I,J,K) or A(I,J).

Array allocation is discussed under Storage Allocation in Section 4.

1.3 STATEMENTS

The FORTRAN-66 elements are combined to form statements. An executable statement performs a calculation or directs control of the program; a non-executable statement provides the compiler with information regarding variable structure, array allocation, storage sharing requirements.

1.4 EXPRESSIONS

An expression is a constant, variable (simple or subscripted), function (section 7.2) or any combination of these separated by operators and parentheses, written to comply with the rules given for constructing a particular type of expression.

There are four kinds of expressions in FORTRAN-66: arithmetic and masking (Boolean) expressions which have numerical values, and logical and relational expressions which have truth values. For each type of expression, there is an associated group of operators and operands.

2. ARITHMETIC EXPRESSION AND REPLACEMENT STATEMENTS

2.1 ARITHMETIC REPLACEMENT STATEMENTS

The general form of the arithmetic replacement statement (arithmetic statement) is A=E, where E is an arithmetic expression and A is any variable name, simple or subscripted. The operator = means that A is replaced by the value of the evaluated expression, E, with conversion for mode if necessary.

2.2 ARITHMETIC EXPRESSIONS

An arithmetic expression can contain the following operators:

OPERATOR	*MEANING*
+	addition
−	subtraction
*	multiplication
/	division
**	exponentiation

Operands are:	Constants
	Variables (simple or subscripted)
	Functions (see Chapter 7)

Examples:

```
A
3.141592
B+16.8946
(A−B(I,J+K) )
G*C(J)+4.1/(Z(I+J,3*K) )*SINF(V)
(Q+V(M,MAXF(A,B) )*Y**2)/(G*H−F(K+3) )
−C+D(I,J)*13.627
```

Any variable (with or without subscripts), constant, or function is an arithmetic expression. These entities may be combined by using the arithmetic operators to form algebraic expressions.

Rules:

1. An arithmetic expression may not contain adjacent arithmetic operators: X op op Y
2. If X is an expression, then (X), ((X)), et cetera, are expressions.
3. If X, Y are expressions, then the following are expressions:

   X+Y   X/Y

   X−Y   X*Y
4. Expressions of the form X**Y and X**(−Y) are legitimate, subject to the restrictions in section 2.3.
5. The following forms of implied multiplication are permitted:

constant (. . .)	implies constant* (. . .)
(. . .) (. . .)	implies (. . .)*(. . .)
(. . .) constant	implies (. . .)*constant
(. . .) variable	implies (. . .)*variable

2.2.1 ORDER OF EVALUATION

The hierarchy of arithmetic operation is:

**	exponentiation	class 1
/	division	class 2
*	multiplication	
+	addition	class 3
−	subtraction	

In an expression with no parentheses or within a pair of parentheses, in which unlike classes of operators appear, evaluation proceeds in the above order. In those expressions where operators of like classes appear, evaluation proceeds from left to right. For example, A**B**C is evaluated as (A**B)**C.

In parenthetical expressions within parenthetical expressions, evaluation begins with the innermost expression. Parenthetical expressions are evaluated as they are encountered in the left to right scanning process.

When writing an integer expression, it is important to remember not only the left to right scanning process, but also that dividing an integer quantity by an integer quantity always yields a truncated result; thus 11/3=3. The expression I*J/K may yield a different result than the expression J/K*I. For example, 4*3/2=6 but 3/2*4=4.

Examples:

In the following examples, R indicates an intermediate result in evaluation:

A**B/C+D*E*F−G is evaluated:

A**B	$\rightarrow R_1$	
R_1/C	$\rightarrow R_2$	
D*E	$\rightarrow R_3$	
R_3*F	$\rightarrow R_4$	
R_4+R_2	$\rightarrow R_5$	
R_5−G	$\rightarrow R_6$	Evaluation completed

A**B/(C+D)*(E*F−G) is evaluated:

A**B	$\rightarrow R_1$	
C+D	$\rightarrow R_2$	
E*F−G	$\rightarrow R_3$	
R_1/R_2	$\rightarrow R_4$	
R_4*R_3	$\rightarrow R_5$	Evaluation completed

If the expression contains a function, the function is evaluated first.

H(13)+C(I,J+2)*(COSF(Z))**2 is evaluated:

COSF(Z)	$\rightarrow R_1$	
R_1**2	$\rightarrow R_2$	
R_2*C(I,J+2)	$\rightarrow R_3$	
R_3+H(13)	$\rightarrow R_4$	Evaluation completed

The following is an example of an expression with embedded parentheses.

A*(B+((C/D)−E)) is evaluated:

C/D	$\rightarrow R_1$	
R_1−E	$\rightarrow R_2$	
R_2+B	$\rightarrow R_3$	
R_3*A	$\rightarrow R_4$	Evaluation completed

A*(SINF(X)+1.)−Z/(C*(D−(E+F))) is evaluated:

SINF(X)	$\rightarrow R_1$	
R_1 + 1	$\rightarrow R_2$	
E + F	$\rightarrow R_3$	
D − R_3	$\rightarrow R_4$	
C * R_4	$\rightarrow R_5$	
Z/R_5	$\rightarrow R_6$	
A * R_2	$\rightarrow R_7$	
R_7 − R_5	$\rightarrow R_8$	Evaluation completed

2.3 MIXED MODE ARITHMETIC EXPRESSIONS

FORTRAN-66 permits full mixed mode arithmetic. Mixed mode arithmetic is accomplished through the special library subroutines. In the 6600 computer system, these routines include complex arithmetic. The five standard operand types are complex, double, real, integer, and logical.

Mixed mode arithmetic is completely general; however, most applications will probably mix operand types, real and integer, real and double, or real and complex. The following rules establish the relationship between the mode of an evaluated expression and the types of the operands it contains.

Rules:

1. The order of dominance of the standard operand types within an expression from highest to lowest is:

   COMPLEX
   DOUBLE
   REAL
   INTEGER
   LOGICAL

2. The mode of an evaluated arithmetic expression is referred to by the name of the dominant operand type.

3. In expressions of the form A**B, the following rules apply:

   - B may be negative in which case the form is: A**(−B).
   - For the standard types (except logical) the mode/type relationships are:

Type A \ Type B	I	R	D	C
I	I	R	D	C
R	R	R	D	C
D	D	D	D	C
C	C	C	C	C

mode of A**B

For example, if A is complex and B is real, the mode of A**B is complex.

2.3.1 EVALUATION

1. Given A, B type real; I, J type integer. The mode of expression A*B−I+J will be real because the dominant operand is type real. It is evaluated:

   Convert I to real

   Convert J to real

   $A*B \rightarrow R_1$ real

   $R_1 - I \rightarrow R_2$ real

   $R_2 + J \rightarrow R_3$ real Evaluation completed

2. The use of parentheses may change the evaluation. A,B,I,J are defined as above. A*B−(I−J) is evaluated:

   $I - J \rightarrow R_1$ integer

   Convert R_1 to real $\rightarrow R_2$

   $A*B \rightarrow R_3$ real

   $R_3 - R_2 \rightarrow R_4$ real Evaluation completed

3. Given C1,C2 type complex; A1,A2 type real. The mode of expression A1*(C1/C2)+A2 is complex because its dominant operand is type complex. It is evaluated:

   $C1/C2 \rightarrow R_1$ complex

   Convert A1 to complex

   Convert A2 to complex

   $A1*R_1 \rightarrow R_2$ complex

   $R_2 + A2 \rightarrow R_3$ complex Evaluation completed

4. Consider the expression C1/C2+(A1−A2) where the operands are defined as in 3 above. It is evaluated:

   $A1 - A2 \rightarrow R_1$ real

   Convert R_1 to complex $\rightarrow R_2$

   $C1/C2 \rightarrow R_3$ complex

   $R_3 + R_2 \rightarrow R_4$ complex Evaluation completed

5. Mixed mode arithmetic with all standard types is illustrated by this example.

   Given: C complex

   D double

   R real

   I integer

   L logical

   and the expression C*D+R/I−L

The dominant operand type in this expression is type complex; therefore, the evaluated expression is of mode complex. It is evaluated:

Convert D,R,I, and L to complex

$R/I \rightarrow R_1$ complex

$C*D \rightarrow R_2$ complex

$R_1 - L \rightarrow R_3$ complex

$R_2 + R_3 \rightarrow R_4$ complex Evaluation completed

If the same expression is rewritten with parentheses as C*D+(R/I−L) the evaluation proceeds:

Convert I to real

Convert L to real

$R/I \rightarrow R_1$ real

$R_1 - L \rightarrow R_2$ real

Convert D to complex

Convert R_2 to complex $\rightarrow R_3$

$C*D \rightarrow R_4$ complex

$R_4 + R_3 \rightarrow R_5$ complex Evaluation completed

2.4 MIXED MODE REPLACEMENT STATEMENT

The mode of an evaluated expression is determined by the type of the dominant operand. This, however, does not restrict the types that identifier A may assume. An expression of complex mode may replace A even if A is type real. The following chart shows the A, E relationship for all the standard modes.

ARITHMETIC REPLACEMENT STATEMENT A = E

A is an Identifier E is an Arithmetic Expression

$\phi(f)$ is the Evaluated Arithmetic Expression

Mode of $\phi(f)$ / *TYPE of A*	*Complex*	*Double*	*Real*	*Integer*
Complex	Store real & imaginary parts of $\phi(f)$ in real & imaginary parts of A.	Round $\phi(f)$ to real. Store in real part of A. Store zero in imaginary part of A.	Store $\phi(f)$ in real part of A. Store zero in imaginary part of A.	Convert $\phi(f)$ to real & store in real part of A. Store zero in imaginary part of A.
Double	Discard imaginary part of $\phi(f)$ & replace it with ± 0 according to real part of $\phi(f)$.	Store $\phi(f)$ (most & least significant parts) in A (most & least significant parts).	If $\phi(f)$ is $\pm$ affix ± 0 as least significant part. Store in A, most & least significant parts.	Convert $\phi(f)$ to real. Fill out least significant half with binary zeros or ones accordingly as sign of $\phi(f)$ is plus or minus. Store in A, most and least significant parts.
Real	Store real part of $\phi(f)$ in A. Imaginary part is lost.	Round $\phi(f)$ to real & store in A. Least significant part of $\phi(f)$ is lost.	Store $\phi(f)$ in A.	Convert $\phi(f)$ to real. Store in A.
Integer	Truncate real part of $\phi(f)$ to INTEGER. Store in A. Imaginary part is lost.	Truncate $\phi(f)$ to INTEGER & store in A.	Truncate $\phi(f)$ to INTEGER. Store in A.	Store $\phi(f)$ in A.
Logical	If real part of $\phi(f) \neq 0$, $1 \rightarrow A$. If real part of $\phi(f) = 0$, $0 \rightarrow A$.	If $\phi(f) \neq 0$, $1 \rightarrow A$. If $\phi(f) = 0$, $0 \rightarrow A$.	Same as for double at left.	Same as for double at left.

When all of the operands in the expression E are of type logical, the expression is evaluated as though all the logical operands were integers.

For example, if L_1, L_2, L_3, L_4 are logical variables, R is a real variable, and I is an integer variable, then

$$I = L_1 * L_2 + L_3 - L_4$$

will be evaluated as though the L_i were all integers (0 or 1) and the resulting value will be stored, as an integer, in I.

$$R = L_1 * L_2 + L_3 - L_4$$

is evaluated as stated above, but the result is converted to a real (a floating point quantity) before it is stored in R.

Examples:

Given:	C_i , A_1	complex
	D_i , A_2	double
	R_i , A_3	real
	I_i , A_4	integer
	L_i , A_5	logical

$A_1=C_1*C_2-C_3/C_4$
$(.905, 15.393)=$
$(4.4,2.1)*(3.0,2.0)-(3.3,6.8)/(1.1,3.4)$

The mode of the expression is complex. Therefore, the result of the expression is a two-word, floating point quantity. A_1 is type complex and the result replaces A_1.

$A_3=C_1$ $\quad$ $4.4000+000=(4.4,2.1)$

The mode of the expression is complex. The type of A_3 is real; therefore, the real part of C_1 replaces A_3.

$A_3=C_1*(0.,-2)$ $\quad$ $4.2000+000=(4.4,2.1)*(0.,-2)$

The mode of the expression is complex. The type of A_3 is real.

$A_4=R_1/R_2*(R_3-R_4)+I_1-(I_2*R_5)$
$3=8.4/4.2*(3.1-2.1)+14-(1*2.3)$

The mode of the expression is real. The type of A_4 is integer; the result of the expression evaluation, a real, will be converted to an integer replacing A_4.

$A_2=D_1**2*(D_2+(D_3*D_4))+(D_2*D_1*D_2)$
$4.96800000000000000000000+001=$
$2.0D**2*(3.2D+(4.1D*1.0D))+(3.2D*2.0D*3.2D)$

The mode of the expression is double. The type of A_2 is double; the result of the expression evaluation, a double precision floating quantity, replaces A_2.

$A_5=C_1*R_1-R_2+I_1$
$1=(4.4,2.1)*8.4-4.2+14$

The mode of the expression is complex. Since A_5 is type logical, an integer 1 will replace A_5 if the real part of the evaluated expression is not zero. If the real part is zero, zero replaces A_5.

3. LOGICAL/RELATIONAL AND MASKING EXPRESSIONS AND REPLACEMENT STATEMENTS

The general form of the logical/relational replacement statement is L=E, where L is a variable of type logical and E may be a logical, relational, or arithmetic expression.

3.1 LOGICAL EXPRESSION

A logical expression has the general form

O_1 op O_2 op O_3 . . .

The terms O_i are logical variables, arithmetic expressions or relational expressions, and op is the logical operator .AND. indicating conjunction or .OR. indicating disjunction.

The logical operator .NOT. indicating negation appears in the form:

.NOT. O_1

The value of a logical expression is either true or false.

When an arithmetic expression appears as a term of a logical replacement statement, the value of the expression is examined. If the value is non-zero, the term has the value TRUE. If the value is equal to zero, the term has the value FALSE.

Logical expressions are generally used in logical IF-statements. (See section 5.3)

Rules:

1. The hierarchy of logical operations is:

First	.NOT.
then	.AND.
then	.OR.

2. A logical variable or a relational expression is, in itself, a logical expression. If L_1, L_2 are logical expressions, then

   .NOT. L_1

   L_1 .AND. L_2

   L_1 .OR. L_2

   are logical expressions. If L is a logical expression, (L), ((L)) are logical expressions.

3. If L_1, L_2 are logical expressions and op is .AND. or .OR. then, L_1 op op L_2 is never legitimate.

4. .NOT. may appear in combination with .AND. or .OR. only as follows:

   .AND. .NOT.

   .OR. .NOT.

   .AND. (.NOT. . . .)

   .OR. (.NOT. . . .)

   .NOT. may appear with itself only in the form .NOT. (.NOT.(.NOT. . . .

5. If L_1, L_2 are logical expressions, the logical operators are defined as follows:

.NOT. L_1	is false only if L_1 is true
L_1 .AND. L_2	is true only if L_1, L_2 are both true
L_1 .OR. L_2	is false only if L_1, L_2 are both false

Incorrect usages such as the following are not permitted:

Q.NOT. .OR.R

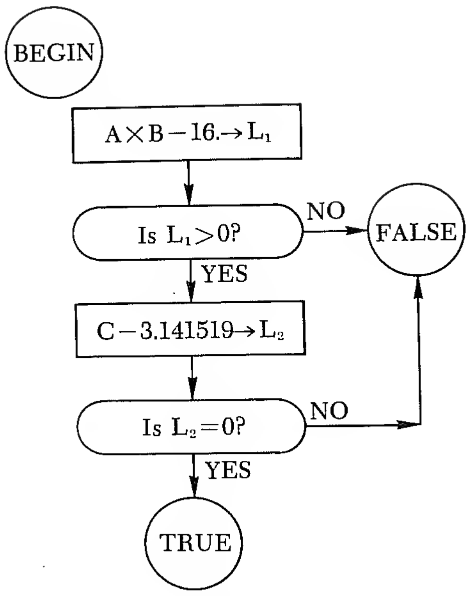
C.AND. .NOT. .NOT.B

The last expression is permissible in the form C.AND. .NOT.(.NOT.B)

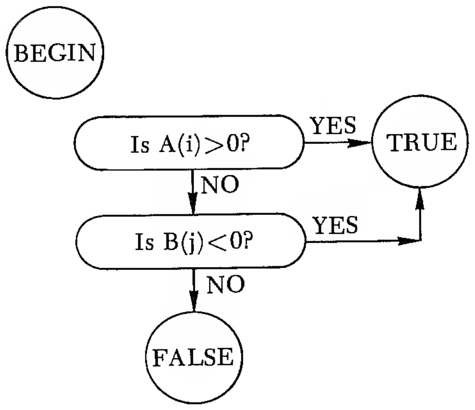
Examples:

Logical Expressions

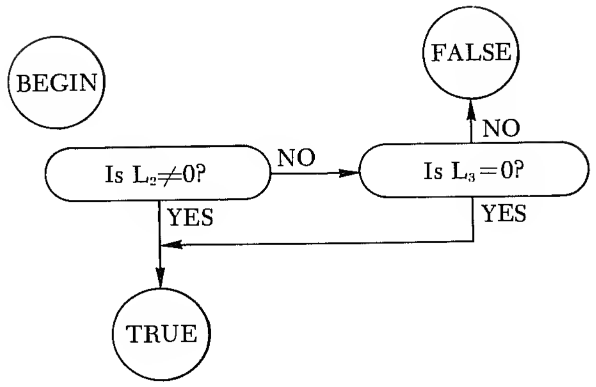
{The product A*B greater than 16.} .AND. {C equals 3.141519}
A*B .GT. 16. .AND. C .EQ. 3.141519


BEGIN
$A \times B - 16. \rightarrow L_1$
Is $L_1 > 0$?
NO
FALSE
YES
$C - 3.141519 \rightarrow L_2$
Is $L_2 = 0$?
NO
YES
TRUE

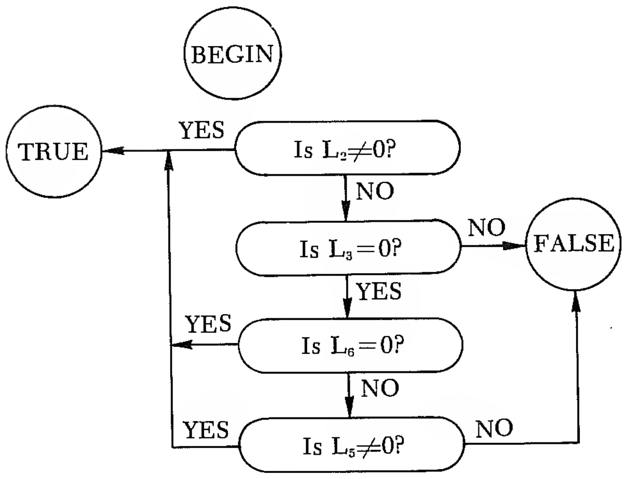
{A(I) greater than 0} .OR. {B(J) less than 0}
A(I).GT.0.OR.B(J).LT.0



In the two examples below,
all L_i are of TYPE LOGICAL
(L2.OR...NOT.L3)



L2.OR..NOT.L3
.AND.(.NOT.L6.OR.L5)



3.2 RELATIONAL EXPRESSION

A relational expression has the form:

q_1 op q_2

The q's are arithmetic expressions; op is an operator belonging to the set:

Operator	*Meaning*
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to
.LT.	Less than
.LE.	Less than or equal to

A relation is true if q_1 and q_2 satisfy the relation specified by op. A relation is false if q_1 and q_2 do not satisfy the relation specified by op.

Relations are evaluated as illustrated in the relation, p .EQ. q. This is equivalent to the question, does $p-q=0$?

The difference is computed and tested for zero. If the difference is zero, the relation is true. If the difference is not zero, the relation is false. Relational expressions are converted internally to arithmetic expressions according to the rules of mixed mode arithmetic. These expressions are evaluated and compared with zero to determine the truth value of the corresponding relational expression. When expressions of mode complex are tested for zero, only the real part is used in the comparison.

Rules:

1. The permissible forms of a relation are:

q	(where a non-zero value is true and a zero value is false)
q_1 op q_2	
q_1 op q_2 .AND. q_2 op q_3	

2. The evaluation of a relation of the form q_1 op q_2 is from left to right. The relations q_1 op q_2, q_1 op (q_2), (q_1) op q_2, (q_1) op (q_2) are equivalent.

Examples:

```
A .GT. 16.                   R(I) .GE. R(I−1)
R−Q(I)*Z .LE. 3.141592       K .LT. 16
B−C .NE. D+E                 I .EQ. J(K)
```

3.3 MASKING REPLACEMENT STATEMENT

The general form of the masking replacement statement is M = E. The masking statement is distinquished from the logical statement in the following ways.

1. The type of M must be real or integer.

2. All operands in the expression E must be type real or integer. E may contain functions as well as variable or constant operands.

Examples:

Given: All variables of type real or integer.

```
A(I)   = B .OR. .NOT. C(I)
B      = D .AND. Q
C(I,J) = .NOT. Z(K) .AND. (Q1 .OR.
           .NOT. Q2)
TEST = CELESTE .AND. 7HECLIPSE
AB     = D .OR. FUNC (X,T)
```

3.4 MASKING EXPRESSIONS

In a FORTRAN-66 masking expression 60-bit arithmetic is performed bit-by-bit on the operands within the expression.

Although the masking operators are identical in appearance to the logical operators, their meanings are different. They are listed accordingly to hierarchy, and the following definitions apply:

.NOT.	complement the operand
.AND.	form the bit-by-bit logical product of two operands
.OR.	form the bit-by-bit logical sum of two operands

The operations are described below.

p	*v*	*p .AND. v*	*p .OR. v*	*.NOT. p*
1	1	1	1	0
1	0	0	1	0
0	1	0	1	1
0	0	0	0	1

Rules:

1. Let B_i be masking expressions, variables or constants whose types are real or integer. Then the following are masking expressions.

   .NOT. B_1 B_1 .AND. B_2 B_1 .OR. B_2

2. If B is a masking expression, then (B), ((B)) are masking expressions.

3. .NOT. may appear with .AND. or .OR. only as follows:

   .AND. .NOT.

   .OR. .NOT.

   .AND. (.NOT. . . .)

   .OR. (.NOT. . . .)

4. Masking expressions of the following forms are evaluated from left to right.

   2 A. .AND. B .AND. C . . .

   A .OR. B. OR. C . . .

5. Masking expressions may contain logical operands, parenthetical arithmetic expressions and statement functions.

Examples:

A_1	77770000000000000000	octal constant
A_2	00000000000077777777	octal constant

B	00000000000000001763	octal form of integer constant
C	20045000000000000000	octal form of real constant

.NOT. A_1	is 00077777777777777777
A_1 .AND. C	is 20040000000000000000
A_1 .AND. .NOT. C	is 57730000000000000000
B .OR. .NOT. A_2	is 77777777777700001763

3.5 MULTIPLE REPLACEMENT STATEMENTS

The multiple replacement statement is a generalization of the replacement statements discussed earlier in this and the previous chapter, and its form is:

$$\psi_n = \psi_{n-1} = \ldots = \psi_2 = \psi_1 = \text{expression}$$

The expression may be arithmetic, logical or masking. The ψ_1 are variables subject to the following restrictions:

Arithmetic or Logical Statement: ψ_1 = ZAP

If ZAP is logical or arithmetic and:

If the variable ψ_1 is type complex, double, real, or integer, then ψ_1 = ZAP is an arithmetic statement.

If the variable ψ_1 is type logical, then ψ_1 = ZAP is a logical statement.

Masking Statement: ψ_1 = ZAP

If ZAP is a masking expression, ψ_1 must be a type real or integer variable only.

The remaining $n-1 \Psi i$ may be variables of any type and the multiple replacement statement replaces each of the variables Ψi $(2 \leqq i \leqq n)$ in turn with the value of $\Psi i-1$ in a manner analogous to that employed in mixed mode arithmetic statements.

Examples:

A	real
E,F	complex
G	double
I	integer
K	logical

The numbers in the examples represent the evaluations of expressions.

```
A = G = 3.14159265358979323846 26D
        3.1415926535897932384626D→G

        3.141592654              →A

I = A = 4.6                4.6→A
                           4  →I

A = I = 4.6                4  →I
                           4.0→A

I = A = E = (10.2,3.0)     10.2→E  real
                            3.0→E
                                imaginary
                           10.2→A
                           10  →I

F = A = I = E = (13.4, 16.2)  13.4→E  real
                              16.2→E
                                  imaginary
                              13  →I
                              13.0→A
                              13.0→F  real
                               0.0→F
                                  imaginary

K= I = −14.6               −14 →I
                             1 →K

I = K= −14.6                 1 →K
                             1 →I
```

4. TYPE DECLARATIONS AND STORAGE ALLOCATIONS

4.1 TYPE DECLARATIONS

The TYPE declaration provides the compiler with information on the structure of variable and function identifiers. There are five standard variable types which are declared by one of the following statements:

Statement	*Characteristics*
TYPE COMPLEX List	2 words/element Floating point
TYPE DOUBLE List	2 words/element Floating point
TYPE REAL List	1 word/element Floating point
TYPE INTEGER List	1 word/element Integer
TYPE LOGICAL List	1 word/element Logical (non-dimensioned)
	1 bit/element, 32 bits/word Logical (dimensioned)

A list is a string of identifiers separated by commas; subscripts are not permitted. An example of a list is:

A, B1, CAT, D36F, EUPHORIA

Rules:

1. The TYPE declaration is non-executable and must precede the first executable statement in a given program.
2. An identifier may not be declared in two or more TYPE declarations.
3. An identifier not declared in a TYPE statement is an integer if the first letter of the identifier is I, J, K, L, M, N; for any other letter, it will be real.
4. If TYPE, DIMENSION or COMMON appear together, the order is immaterial.

Examples:

```
TYPE COMPLEX A147, RIGGISH, AT1LL2
TYPE DOUBLE  TEEPEE, B2BAZ
TYPE REAL    EL, CAMINO, REAL, IDE63
TYPE INTEGER QUID, PRO, QUO
TYPE LOGICAL GEORGE6
```

4.2 DIMENSION

A subscripted variable represents an element of an array of variables. Storage may be reserved for arrays by the non-executable statements DIMENSION or COMMON.

The standard form of the DIMENSION statement is

DIMENSION $V_1, V_2, \ldots, V_n$

The variable names, V_i, may have 1, 2, or 3 integer constant subscripts separated by commas, as in SPACE (5, 5, 5,). Under certain conditions within subprograms only, the subscripts may be integer variables. This is explained in section 6.11.

The number of computer words reserved for a given array is determined by the product of the subscripts in the subscript string, and the type of the variable.

In the statements

```
TYPE COMPLEX HERCULES
DIMENSION HERCULES (10, 20)
```

the number of elements in the array HERCULES is 200. Two words are used to store a complex element; therefore, the number of computer words reserved is 400. The argument is the same for double precision. For reals and integers the number of words in an array equals the number of elements in the array.

Rules:

1. The DIMENSION statement is non-executable and must precede the first executable statement in a given program.
2. An identifier may not be declared in two or more DIMENSION statements.
3. If TYPE, DIMENSION or COMMON appear together, the order is immaterial.
4. For any given dimensional variable, the dimensions may be declared either in a COMMON statement or in a DIMENSION statement. If declared in both, those of the DIMENSION statement override those declared in the COMMON statement.
5. Any number of DIMENSION statements may appear in a program section.
6. A zero subscript is treated as a one.

4.2.1 VARIABLE DIMENSIONS

When an array identifier and its dimensions appear as formal parameters in a function or subroutine, the dimensions may be assigned through the actual parameter list accompanying the function reference or subroutine call. The dimensions must not exceed the maximum array size specified by the DIMENSION statement in the calling program. See section 6.11 for details and examples.

4.3 COMMON

A program may contain or call subprograms. Areas of common information may be specified by the statement:

COMMON/I_1/List/I_2/List . . .

I is a common block identifier. It may be blank; or consist of up to 8 characters. If the first character is a number, then all characters in the identifier must be numbers. The following are common identifiers:

/AZ13/	/3597/
/CAVEAT/	/ /
/0/	/X/

List is composed of simple variable identifiers and array identifiers (subscripted or non-subscripted). If a non-subscripted array name appears in the list, the dimensions are defined by the DIMENSION statement in that program.

Dimensions for arrays may also be given in the COMMON statement when a subscript string appears with the identifier. If dimensions are given in both, those in the DIMENSION statement are used.

Examples:

```
COMMON A, B, C
COMMON/ / A, B, C, D
COMMON/BLOCK1/A, B/1234/C(10),
   D(10,10),E(10,10,10)
COMMON/BLOCKA/D(15), F(3,3),
   GOSH(2, 3, 4), Q1
```

If a subprogram does not use all of the locations reserved in a common block, unused variables may be necessary in the COMMON statement to insure proper correspondence of common areas.

```
MAIN PROG      COMMON/SUM/A, B, C
SUB PROG       COMMON/SUM/E, F, G
```

In the above example, assume only the variables E and G are used in the subprogram. The unused variable F is necessary to space over the area reserved by B.

Rules:

1. COMMON is non-executable and must precede the first executable statement in the program. Any number of COMMON statements may appear in a program section.
2. If TYPE, DIMENSION or COMMON appear together, the order is immaterial.
3. A variable in COMMON may not be equated to any other variable in COMMON.
4. Common block identifiers are used only for block identification within the compiler; they may be used elsewhere in the program as other kinds of identifiers.
5. An identifier in one common block may not appear in another common block.
6. For any given dimensional variable, the dimensions may be declared either in a COMMON statement or in a DIMENSION statement. If declared in both, those of the DIMENSION statement override those declared in the COMMON statement.
7. At the beginning of program execution, the contents of the common area are undefined unless specified by a DATA statement.

4.4 COMMON BLOCKS

The COMMON statement provides the programmer with a means of reserving blocks of storage area that are referenced by more than one subprogram. Data may be stored in common blocks by the DATA statement and are made available to any subprogram using the appropriate block.

4.4.1 BLOCK LENGTH

The length of a common block in computer words is determined from the number and type of the list identifiers. In the following statement, the length of the common block A is 12 computer words. The origin of the common block is Q(1). Q and R are real variables and S is complex.

COMMON/A/Q(4), R(4), S(2)

	block A	
origin	Q(1)	
	Q(2)	
	Q(3)	
	Q(4)	
	R(1)	
	R(2)	
	R(3)	
	R(4)	
	S(1)	real part
	S(1)	imaginary part
	S(2)	real part
	S(2)	imaginary part

Example:

```
MAIN PROG
COMPLEX C
COMMON/TEST/C(20)/36/A,B,Z
     .
     .
     .
```

The length of TEST is 40 computer words.
The subprogram may rearrange the allocation of words as in:

```
SUB PROG 1
COMMON/TEST/A(10),G(10),K(10)
TYPE COMPLEX A
     .
     .
     .
```

The length of TEST is 40 words. The first 10 elements (20 words) of the block, represented by A, are complex elements. Array G is the next 10 words, and array K is the last 10 words. Within the subprogram, elements of G will be treated as floating point quantities; elements of K will be treated as integer quantities.

The length of a COMMON block must not be changed by the subprograms using the block. The identifiers used within the block may differ as shown above.

The following arrangements are equivalent:

```
TYPE DOUBLE A      }   { COMMON A(10)
DIMENSION A(10)    }   { TYPE DOUBLE A
COMMON A           }   {
```

The label of a COMMON block is used only for block identification. The following is permissible:

```
COMMON /A/A(10)/B/B(5,5) /C/C (5,5,5)
```

4.5 EQUIVALENCE

The EQUIVALENCE statement permits variables to share locations in storage. The general form is:

```
EQUIVALENCE (A,B,...), (A1,B1,...),...
```

(A,B,...) is an equivalence group of two or more simple or singly subscripted variable identifiers. A multiply subscripted variable can be represented only by a singly subscripted variable. The correspondence is:

$$A(i,j,k) = A(i + (j-1)\ I + (k-1)IJ)$$

where i,j,k are integer constants: I and J are the integer constants appearing in DIMENSION A (I,J,K). For example, in DIMENSION A(2,3,4), the element A(1,1,2) is represented by A(7).

Examples:

EQUIVALENCE is most commonly used when two or more arrays can share the same storage locations. The lengths may be different or equal.

```
    DIMENSION A(10,10), I(100)
    EQUIVALENCE (A,I)
              .
              .
              .
5   READ 10, A
              .
              .
              .
6   READ 20, I
```

The EQUIVALENCE statement assigns the first element of array A and array I to the same storage location. The READ statement 5 stores the A array in consecutive locations. Before statement 6 is executed all operations using A must be completed as the values of array I are read into the storage locations previously occupied by A.

Rules:

1. EQUIVALENCE is non-executable and must precede the first executable statement in the program or subprogram.

2. If TYPE, DIMENSION, COMMON, or EQUIVALENCE appear together, the order is immaterial.

3. Any variable or array may be made equivalent to any other variable or array. The variable or array may or may not be subscripted in the EQUIVALENCE statement. A given identifier may be repeated in more than one equivalence group, but caution must be observed to avoid circular references. A zero or one subscript is treated as though the subscript were not present.

4. The EQUIVALENCE statement does not rearrange common, but arrays may be defined as equivalent so that the length of the common block is changed. The origin of the common block must not be changed by the EQUIVALENCE statement. The following simple examples illustrate changes in block lengths caused by the EQUIVALENCE statement.

Given: Arrays A and B
Sa = subscript of A
Sb = subscript of B

Examples:

1. A in COMMON, B not in COMMON

   Sb $\leq$ Sa is a permissible subscript

   Sb $>$ Sa is not

```
COMMON/1/A(4)
DIMENSION B(5)
EQUIVALENCE (A(3), B(2))
origin  A(1)
        A(2)     B(1) \
        A(3)     B(2)  |
        A(4)     B(3)  >   length of block 1
                 B(4)  |   (now 6 elements)
                 B(5) /
```

2. B in COMMON, A not in COMMON

   Sa $\leq$ Sb is a permissible subscript

   Sa $>$ Sb is not

```
COMMON/2/B(3)
DIMENSION A(2)
EQUIVALENCE (B(1), A(1))
origin  B(1)     A(1) \
        B(2)     A(2)  >   length of block 2
        B(3)          /    (not increased)
```

3. A, B not in COMMON

   No restrictions on subscript arrangement.

4. A, B both in COMMON

   No subscript arrangement is legal.

4.6 DATA

The programmer may assign constant values to variables in the source program by using the DATA statement either by itself or with a DIMENSION statement. It may be used to store constant values in variables contained in a common block.

DATA(I_1=List), (I_2=List), . . .

I is an identifier representing a simple variable, array name, or a variable with integer constant subscripts or integer variable subscripts.

List contains constants only and has the form

$a_1, a_2, \ldots, k(b_1, b_2, \ldots), c_1, c_2 \ldots$

k is an integer constant repetition factor that causes the parenthetical list following it to be repeated k times.

Rules:

1. DATA is non-executable and must precede the first executable statement in any program or subprogram in which it appears.

2. When DATA appears with TYPE, DIMENSION, COMMON or EQUIVALENCE statements, the order is immaterial.

3. DO loop-implying notation is permissible. This notation may be used for storing constant values in arrays.

```
DIMENSION X(10,10)
DATA (((X(I,J),I = 1,10,2), J = 6,9,3)
                          = 2(5(3.1)))
```

```
DIMENSION GIB (10)
DATA ( (GIB(I),I=1,10)=1. ,2. ,3. ,7(4.32) )
  ARRAY GIB

                    1.
                    2.
                    3.
                    4.32
                    4.32
                    4.32
                    4.32
                    4.32
                    4.32
                    4.32
```

4. Variables in variable dimensioned arrays may not be preset in a DATA statement.

5. Either unsigned constants or constants preceded by a minus sign may be used. Negative octal constants are prefixed with minus signs.

6. There must be a one-to-one correspondence between the identifiers and the list.

```
COMMON / TUP / C(3)
DATA (C=1. , 2, 3)
```

Examples:

1. DATA (LEDA=15), (CASTOR=16.0), (POLLUX=84.0)

```
        LEDA        15
                     .
                     .
                     .
      CASTOR        16.0
                     .
                     .
                     .
      POLLUX        84.0
```

2. DATA (A(1,3) = 16.239)

```
     ARRAY A
        A(1,3)      16.239
```

3. DIMENSION B(10)

```
   DATA (B=77B, -77B, 4(776B, -774B) )
      ARRAY B        77B
                    -77B
                    776B
                   -774B
                    776B
                   -774B
                    776B
                   -774B
                    776B
                   -774B
```

4.7 BANK

In FORTRAN-66 the BANK statement acts as a do-nothing statement.

5. CONTROL STATEMENTS

Program execution normally proceeds from one statement to the statement immediately following it in the program. Control statements can be used to alter this sequence or cause a number of iterations of a program section.

Control may be transferred to an executable statement only; a transfer to a non-executable statement results in a program error.

Iteration control provided by the DO statement causes a predetermined sequence of instructions to be repeated any number of times with the stepping of a simple integer variable after each iteration.

5.1 STATEMENT IDENTIFIERS

Statements are identified by numbers which can be referred to from other sections of the program. A statement number used as a label or tag appears in columns 1 through 5 on the same line as the statement on the coding form. The statement number N may lie in the range $1 \leqq N \leqq 99999$. An identifier up to 5 digits long may occupy any of the first five columns; blanks are squeezed out and leading zeros are ignored. 1, 01, 001, 0001, are identical, (Appendix A).

5.2 GO TO STATEMENTS

Transfer of control is provided by GO TO statements.

5.2.1 UNCONDITIONAL GO TO

GO TO N

This statement causes an unconditional transfer to the statement labeled N; N is a statement identifier.

5.2.2 ASSIGNED GO TO

GO TO m, $(n_1,n_2 \ldots ,n_m)$

This statement acts as a many-branch GO TO. m is an integer variable assigned an integer value n_i in a preceeding ASSIGN statement. The n_i are statement numbers. A parenthetical list need not be present.

The comma after m is optional when the list is omitted. m cannot be the result of a computation. If m is computed, the object code is incorrect.

5.2.3 ASSIGN STATEMENTS

ASSIGN N TO m

This statement is used with the assigned GO TO statement. N is a statement number, m is a simple integer variable.

```
       ASSIGN 10 TO LSWTCH
                .
                .
                .
       GO TO LSWTCH,(5,10,15,20)
```

Control will transfer to statement 10.

5.2.4 COMPUTED GO TO

GO TO $(n_1,n_2, \ldots ,n_m)i$

GO TO $(n_1,n_2 \ldots ,n_m)$, i

This statement acts as a many-branch GO TO, where i is preset or computed prior to its use in the GO TO.

The n_i are statement numbers and i is a simple integer variable. If $i \leqq 1$, a transfer to n_1 occurs; if $i \geqq m$, a transfer to n_m occurs. Otherwise, transfer is to n_i.

For proper operations, i must not be specified by an ASSIGN statement.

```
       ISWITCH = 1
       GO TO (10,20,30),ISWITCH
          .
          .
          .
   10  JSWITCH = ISWITCH + 1
       GO TO (11,21,31),JSWITCH
```

Control will transfer to statement 21.

5.3 IF STATEMENTS

Conditional transfer of control is provided by the one-, two-, and three-branch IF statements, the status of sense lights or switches.

5.3.1 THREE BRANCH IF (ARITHMETIC)

IF (A) n_1,n_2,n_3

A is an arithmetic expression and the n_i are statement numbers. This statement tests the evaluated quantity A and jumps accordingly.

$A < 0$	jump to statement n_1
$A = 0$	jump to statement n_2
$A > 0$	jump to statement n_3

In the test for zero, $+0= -0$. When the mode of the evaluated expression is complex, only the real part is tested for zero.

```
IF(A*B-C*SINF(X) )10,10,20
IF(I)5,6,7
IF(A/B**2)3,6,6
```

5.3.2 TWO BRANCH IF (LOGICAL)

IF(L) n_1,n_2

L is a logical expression. The n_i are statement numbers.

The evaluated expression is tested for true (non-zero) or false (zero). If L is true, jump to statement n_1. If L is false, jump to statement n_2.

```
IF(A .GT. 16. .OR. I .EQ.0)5,10
IF(L)1,2              (L is TYPE LOGICAL)
IF(A*B-C)1,2          (A*B-C is arithmetic)
IF(A*B/C .LE. 14.32)4,6
```

5.3.3 ONE BRANCH IF (LOGICAL)

IF (L)s

L is a logical expression and s is a statement. If L is true (non-zero), execute statement s. If L is false (zero), continue in sequence to the statement following the IF logical.

```
IF (L) GO TO 3 (L is logical)
IF (L) Y=SINF (X) /2
```

5.3.4 SENSE LIGHT

SENSE LIGHT i

The statement turns on the sense light i. SENSE LIGHT 0 turns off all sense lights. i may be a simple integer variable or constant (1 to 60).

IF(SENSE LIGHT i) n_1,n_2

The statement test sense light i. If it is on, it is turned off and a jump occurs to statement n_1. If it is off, a jump occurs to statement n_2. i is a sense light and the n_i are statement numbers. i may be a simple integer variable or constant (1 to 60).

```
IF(SENSE LIGHT 4)10,20
```

5.3.5 SENSE SWITCH

IF(SENSE SWITCH i) n_1,n_2

If sense switch i is set (on), a jump occurs to statement n_1. If it is not set (off), a jump occurs to statement n_2; i may be a simple integer variable or constant. In the 6600, $1 \leqq i \leqq 6$

```
N=5
IF(SENSE SWITCH N) 5,10
```

5.4 FAULT CONDITION STATEMENTS

At execute time, the computer is set to interrupt on operand out of range. A program may be executed with an exit on any fault condition or by optional exits or tests provided by the program (SIPROS parameter).

IF DIVIDE CHECK n_1,n_2

IF DIVIDE FAULT n_1,n_2

The above statements are equivalent. A divide fault occurs following division by zero. The statement checks for this fault; if it has occurred, the indicator is turned off and a jump to statement n_1 takes place. If no fault exists, a jump to statement n_2 takes place.

IF EXPONENT FAULT n_1,n_2

An exponent fault occurs when the result of a real or complex arithmetic operation exceeds the upper limits specified for these types. Results that are less than the lower limits are set to zero without indication. This statement is therefore a test for floating-point overflow only. If the fault occurs, the indicator is turned off, and a jump to statement n_1 takes place. If no fault exists, a jump to statement n_2 takes place.

IF OVERFLOW FAULT n_1,n_2

No overflow condition is provided on 60-bit sums and differences. A jump to statement n_2 is generated for the statement.

5.5 DO STATEMENTS

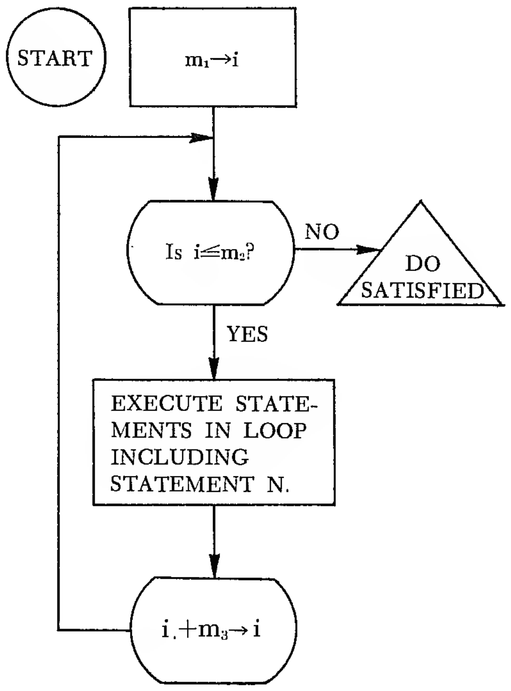
DO n $i=m_1,m_2,m_3$

This statement makes it possible to repeat groups of statements and to change the value of a fixed point variable during the repetition. n is the number of the statement ending the DO loop. i is the index variable (simple integer). The m_i are the indexing parameters; they may be any computable arithmetic expression. The initial value assigned to i is m_1, m_2 is the largest value assigned to i, and m_3 is the amount added to i after each DO loop is executed. If m_3 does not appear, it is assigned the value 1. The values of m_2 and m_3 may be changed at any time.

The DO statement, the statement labeled n, and any intermediate statements constitute a DO loop. Statement n may not be an IF or GO TO statement or another DO statement. See Transmission of Arrays section 7.1.1 and DATA Statement section 4.6 for usage of implied DO loops.

5.5.1 DO LOOP EXECUTION

The initial value of i, m_1, is compared with m_2 before executing the DO loop and, if it does not exceed m_2, the loop is executed. After this step, i is increased by m_3. i is again compared with m_2; this process continues until i exceeds m_2 as shown below. Control then passes to the statement immediately following n, and the DO loop is satisfied.

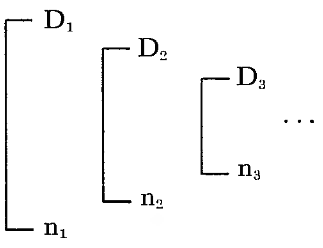


If m_1 exceeds m_2 on the initial entry to the loop, the loop is not executed and control passes to the next statement after statement n.

When the DO loop is satisfied, the index variable i is no longer well defined. If a transfer out of the DO loop occurs before the DO is satisfied, the value of i is preserved and may be used in subsequent statements.

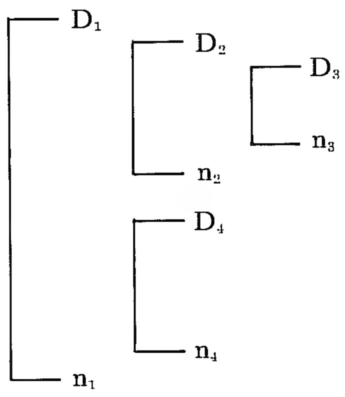
5.5.2 DO NESTS

When a DO loop contains another DO loop, the grouping is called a DO nest. The last statement of a nested DO loop must either be the same as the last statement of the outer DO loop or occur before it. If $D_1,D_2,\ldots D_m$ represent DO statements, where the subscripts indicate that D_1 appears before D_2 appears before D_3, and $n_1,n_2,\ldots,n_m$ represent the corresponding limits of the D_i, then n_m must appear before $n_{m-1}\ldots n_2$ must appear before n_1.

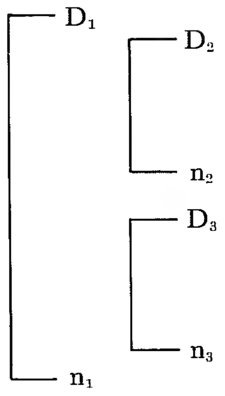


Examples:

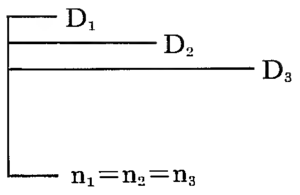
DO loops may be nested in common with other DO loops:



```
DO 1 I = 1,10,2
      .
      .
      .
DO 2 J = 1,5
      .
      .
      .
DO 3 K = 2,8
      .
      .
      .
3 CONTINUE
      .
      .
      .
2 CONTINUE
      .
      .
      .
DO 4 L = 1,3
      .
      .
      .
4 CONTINUE
      .
      .
      .
1 CONTINUE
```



```
DO 100 L = 2,LIMIT
      .
      .
      .
DO 10 I = 1,10
DO 10 J = 1,10
      .
      .
      .
10 CONTINUE
      .
      .
      .
DO 20 K = K1,K2
      .
      .
      .
20 CONTINUE
      .
      .
      .
100 CONTINUE
```



```
DO 5 I = 1,5
DO 5 J = I,10
DO 5 K = J,15
      .
      .
      .
5 CONTINUE
```

5.5.3 DO LOOP TRANSFER

In a DO nest, a transfer may be made from one DO loop into a DO loop that contains it; and a transfer out of a DO nest is permissible.

The special case is transferring out of a nested DO loop and then transferring back to the nest. In a DO nest, if the range of i includes the range of j and a transfer out of the range of j occurs, then a transfer into the range of i or j is permissible.

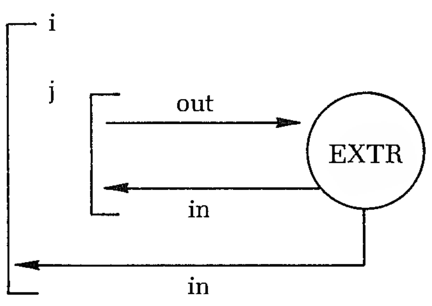
A transfer from the outer DO loop to the last statement should not be made when that statement is also the last statement of an inner DO loop. If such a transfer is made, control will transfer to the inner DO loop.

Example:

```
        DO 10 I=J,K,L
          .
          .
          .
      5 GO to 10
        DO 10 II=JJ,KK,LL
          .
          .
          .
     10 CONTINUE
```

This will cause control to pass the inner DO loop.

In the following diagram, EXTR represents a portion of the program outside of the DO nest.



5.5.4 DO PROPERTIES

1. The indexing parameters m_1,m_2,m_3 may be any executable arithmetic expression.
2. The indexing parameters m_1 and m_2, if variable, may assume positive or negative values, or zero.
3. The values of m_2 and m_3 may be changed during the execution of the DO loop.
4. i is initially m_1. As soon as i exceeds m_2, the loop is terminated.
5. DO loops may be nested 50 deep.

5.6 CONTINUE

CONTINUE

The CONTINUE statement is most frequently used as the last statement of a DO loop to provide a transfer address for IF and GO TO instructions that are intended to begin another repetition of the loop. If CONTINUE is used elsewhere in the source program, it acts as a do-nothing instruction; and control passes to the next sequential program statement.

5.7 PAUSE

PAUSE

PAUSE n

PAUSE or PAUSE n act as do-nothing instructions.

5.8 STOP

STOP

STOP n

STOP or STOP n cause immediate exit to the monitor. n is an arbitrary number, not more than 5 digits in length.

5.9 END

END

END marks the physical end of a program or subprogram. It is executable in the sense that it effects return from a subprogram in the absence of a RETURN.

The END statement may include the name of the program or subprogram which it terminates. This name, however, is ignored.

The END statement is not connected with the dollar sign, which is used as a separator, nor does it make use of the continuation. When punched into a card, it must be the only entry in the card and must be punched within columns 7 through 72.

6. FUNCTIONS AND SUBPROGRAMS

Sets of instructions may be written as independent subroutines or function subprograms which can be referred to by the main program. The mode of a function subprogram is determined by the name of the subroutine in the same manner as variable modes are determined. Subroutine subprogram names are not classified by mode. The program or function name must be unique within that subprogram.

6.1 MAIN PROGRAMS AND SUBPROGRAMS

A main program may be written with or without references to subprograms. In all cases, the first statement must be of the following form where name is an alphanumeric identifier, 1-8 characters. The first character must be alphabetic; the remaining characters may be alphabetic or numeric.

PROGRAM name

A main program may refer to both subroutines and functions which are compiled independently of the main program. A calling program is a main program or subprogram that refers to subroutines and functions.

6.2 FUNCTIONAL SUBPROGRAM

The first statement of function subprograms must have the form:

FUNCTION F $(p_1, p_2, \ldots p_n)$

F is the function name and the p_i are formal parameters. A function name is constructed and its type determined in the same way as a variable identifier. The parameters may be array names, non-subscripted variables, or names of other function or subroutine subprograms. A function, together with its arguments, may be used any place in an expression that a variable identifier may be used.

A function reference is a call upon a computational procedure for the return of a single value associated with the function identifier. This procedure may be defined by a single statement in the program (arithmetic statement function), in the compiler (library function), or in a multi-statement subprogram compiled independently of a main program (function subprogram).

The name of a function subprogram may occur as an operand in an arithmetic statement. The function reference must supply the function with at least one argument and it may contain up to 63. The form of the function reference is:

F $(p_1, p_2, \ldots p_n)$

F is the function name and p_i are function arguments or *actual* parameters. The corresponding arguments appearing with the function name in a function definition are called *formal* parameters. Because formal parameters are local to the subprogram in which they appear, they may or may not correspond to the actual parameters in the calling subprogram.

Type-FUNCTION statements are permitted in addition to the FUNCTION F $(p_1, p_2, \ldots p_n)$ statement where mode is determined by the first character or a TYPE declaration. The alternate forms are:

REAL FUNCTION F$(p_1,p_2, \ldots p_n)$
INTEGER FUNCTION F$(p_1,p_2, \ldots p_n)$
DOUBLE PRECISION FUNCTION
F $(p_1,p_2, \ldots p_n)$
COMPLEX FUNCTION F $(p_1,p_2, \ldots p_n)$
LOGICAL FUNCTION F $(p_1,p_2, \ldots p_n)$

F is the function name, and p_i are formal parameters. The type-FUNCTION statement declares the type of the result returned by the function.

Rules:

1. The type of the function is determined from the naming conventions specified for variables in section 4.1 or from the type-FUNCTION statement.
2. The name of a function must not appear in a DIMENSION statement. The name must appear, however, at least once as any of the following:

   the left-hand identifier of a replacement statement

   an element of an input list

   an actual parameter of a subprogram call
3. No element of a formal parameter list may appear in a COMMON, EQUIVALENCE, DATA, or EXTERNAL statement within the function subprogram.

4. When a formal parameter represents an array, it must be declared in a DIMENSION statement within the function subprogram. If it is not declared, only the first element of the array is available to the function subprogram.

5. In referring to a function subprogram the following forms of the actual parameters are permissible:

   arithmetic expression

   constant or variable, simple or subscripted

   array name

   function reference

   subroutine

When the name of a function subprogram appears as an actual parameter, that name must also appear in an EXTERNAL statement in the calling program. When the name of a function subprogram is used as an actual parameter, it may be in either of two forms:

a. FUNCTION PUSH (ARG1)

b. FUNCTION PUSH (FUNCT,ARG2)

To call these functions, then, the calling program uses, respectively, the forms:

a. A = PUSH (CALC(X))

b. A = PUSH (CALC, X)

When a subroutine appears as an actual parameter, the subroutine name may appear alone or with a parameter list. When a subroutine appears with a parameter list, the subroutine name and its parameters must appear as separate actual parameters:

Level I—Calling program

```
    A = PUSH (DAB,W,X)


    B = PUSH (DABL,Y,Z)
```

Level II—Function statement and subroutine call

```
    FUNCTION PUSH (SUB, ARG1, ARG2)


    CALL SUB (ARG1, ARG2)

    PUSH = - -
    END
```

At some other level of the program, subroutines DAB and DABL are defined.

6. Logical expressions may be actual parameters.

7. Actual and formal parameters must agree in order, number (see rule 5, section 6.7) and type.

8. Functions must have at least one parameter.

6.3 LIBRARY FUNCTIONS

Function subprograms which are used frequently have been written and stored in a reference library and are available to the programmer through the compiler.

FORTRAN-66 contains the standard library functions available in earlier versions of FORTRAN. A list of these functions is in Appendix D. When one appears in the source program, the compiler identifies it as a library function and generates a calling sequence within the object program.

In the absence of a TYPE declaration, the type of the function identifier is determined by its first letter or by the type-FUNCTION statement. However, for standard library functions the modes of the results have been established through usage. The compiler recognizes the standard library functions and associates the established types with the results.

For example, in the function identifier LOGF, the first letter, L, would normally cause that function to return an integer result. This is contrary to established FORTRAN usage. The compiler recognizes LOGF as a standard library function and permits the return of a real result.

6.4 EXTERNAL STATEMENT

When the actual parameter list of a function or subroutine reference contains a function or subroutine name, that name must be declared in an EXTERNAL statement. Its form is:

EXTERNAL identifier$_1$, identifier$_2$, . . .

Identifier$_i$ is the name of a function or subroutine. The EXTERNAL statement must precede the first executable statement of any program in which it appears. When it is used, EXTERNAL always appears in the calling program; it must not be used with arithmetic statement functions.

Examples:

1. *Function Subprogram*

```
FUNCTION GREATER (A,B)
IF (A.GT.B) 10,20
10 GREATER = A−B
   RETURN
20 GREATER = A+B
   END
```

Calling Program Reference

```
Z(I,J)=F1+F2−GREATER(C−D,3.*I/J)
```

2. *Function Subprogram*

```
FUNCTION PHI(ALFA, PHI2)
PHI = PHI2(ALFA)
END
```

Calling Program Reference

```
EXTERNAL SINF
   .
   .
   .
C = D−PHI(Q(K),SINF)
```

From its call in the main program, the formal parameter ALFA is replaced by Q(K), and the formal parameter PHI2 is replaced by SINF. PHI will be replaced by the sine of Q(K).

3. *Function Subprogram*

```
FUNCTION PSYCHE (A,B,X)
CALL X
PSYCHE = A/B*2.*(A−B)
END
```

Function Sumprogram Reference

```
EXTERNAL EROS
   .
   .
   .
R = S−PSYCHE (TLIM, ULIM, EROS)
```

In the function subprogram, TLIM, ULIM replaces A,B. The CALL X is a call to a subroutine named EROS. EROS appears in an EXTERNAL statement so that the compiler recognizes it as a subroutine name rather than a variable identifier.

4. *Function Subprogram*

```
FUNCTION AL(W,X,Y,Z)
CALL W(X,Y,Z)
AL = Z**4
RETURN
END
```

Function Subprogram Reference

```
EXTERNAL SUM
   .
   .
   .
G = AL(SUM,E,V,H)
```

In the function subprogram the name of the subroutine (SUM) and its parameters (E,V,H) replace W and X,Y,Z. SUM appears in the EXTERNAL statement so that the compiler treats it as a subroutine name rather than a variable identifier.

6.5 STATEMENT FUNCTIONS

Statement functions are defined when used as an operand in a single arithmetic or logical statement in the source program and apply only to the particular program or subprogram in which the definition appears. They have the form

$$F\,(p_1,p_2,\ldots p_n) = E \qquad 1 \leqq n \leqq 63$$

F is the function name, p_i are the formal parameters, and E is an expression.

Rules:

1. The function name must not appear in a DIMENSION, EQUIVALENCE, COMMON or EXTERNAL statement.

2. The formal parameters usually appear in the expression E. When the statement function is executed, formal parameters are replaced by the corresponding actual parameters of the function reference. Each of the formal parameters may be TYPE REAL or INTEGER only, but they may not be declared in a TYPE statement. Each of the actual parameters may be any computable arithmetic expression, but there must be agreement in order, number and type between the actual and formal parameters. Formal parameters must be simple variables.

3. E may be an arithmetic or logical expression.
4. E may contain subscripted variables.
5. The expression E may refer to library functions, previously defined statement function and function subprograms.
6. All statement functions must precede the first executable statement of the program or subprogram, but they must follow all declarative statements (DIMENSION, TYPE, et cetera).

Examples:

```
TYPE COMPLEX Z
Z(X,Y) = (1.,0.)*EXPF(X)*COSF(Y)
            + (0.,1.)*EXPF(X)*SINF(Y)
```

This arithmetic statement function computes the complex exponential $Z(x,y) = e^{x+iy}$.

6.6 SUBROUTINE PROGRAM

A reference to a subroutine is a call upon a computational procedure. This procedure may return none, one or more values. No value is associated with the name of the subroutine, and the subroutine must be called by a CALL statement.

The first statement of subroutine subprograms must have the form:

SUBROUTINE S
or
SUBROUTINE S $(p_1,p_2,\ldots p_n)$ $1 \leq n \leq 63$

S is the subroutine name which follows the rules for variable identifiers, and p_i are the formal parameters which may be array names, non-subscripted variables, or names of other function or subroutine subprograms.

Rules:

1. The name of the subroutine may not appear in any declarative statement (TYPE, DIMENSION) in the subroutine.
2. No element of a formal parameter list may appear in a COMMON, EQUIVALENCE, DATA, or EXTERNAL statement within the subroutine subprogram.
4. When a formal parameter represents an array, it must be declared in a DIMENSION statement within the subroutine. If it is not declared, only the first element of the array is available to the subroutine.

6.7 CALL

The executable statement in the calling program for referring to a subroutine subprogram is of the form:

CALL S
or
CALL S $(p_1,p_2,\ldots p_n)$ $1 \leq n \leq 63$

S is the subroutine name, and p_i are the actual parameters. The CALL statement transfers control to the subroutine. When a RETURN or END statement is encountered in the subroutine, control is returned to the next executable statement following the CALL in the calling program. If the CALL statement is the last statement in a DO, looping continues until satisfied. Subprograms may be called from a main program or from other subprograms. Any subprogram called, however, may not call the calling program. That is, if program A calls subprogram B, subprogram B may not call program A. Furthermore, a program or subprogram may not call itself.

Rules:

1. The subroutine returns values through formal parameters which are substituted for actual parameters or through common variables.
2. The subroutine name may not appear in any declarative statement (TYPE, DIMENSION, et cetera). No value is associated with its name.
3. In the subroutine call, the following forms of actual parameters are permissible:

   arithmetic expression
   constant or variable, simple or subscripted
   array name
   function reference
   subroutine or function name
   logical expression

When the name of a function subprogram appears as an actual parameter, that name must also appear in an EXTERNAL statement in the calling program. When the name of a function subprogram is used as an actual parameter, it may be in either of two forms:

a. FUNCTION POSH (ARG3)
b. FUNCTION POSH (FUNCT, ARG4)

To call these functions, then, the calling program uses, respectively, the forms:

a. B = POSH (CALC(X))

b. B = POSH (CALC,X)

When a subroutine appears as an actual parameter, the subroutine name may appear alone or with a parameter list.

When a subroutine appears with a parameter list, the subroutine name and its parameters must appear as separate actual parameters.

Level I—Calling program

```
      .
      .
A = PULL (DIS,A,B)
      .
      .
B = PULL (DISP,C,D)
```

Level II—Function statement and subroutine call

```
FUNCTION PULL (SUB,ARG1,ARG2)
      .
      .
CALL SUB (ARG1, ARG2)
      .
      .
PULL = - -
END
```

Level III—Subroutine definition

```
SUBROUTINE DIS (DUMMY 1, DUMMY 2)
      .
      .
END
SUBROUTINE DISP (DUMMY 3, DUMMY 4)
      .
      .
END
```

4. Because formal parameters are local to the subroutine in which they appear, they may be the same as names appearing outside the subroutine.

5. Actual and formal parameters must agree in order, type and, for the first call of the subprogram, in number (example 4).

6. Logical expressions may be actual parameters.

Examples:

1. *Subroutine Subprogram*

```
SUBROUTINE BLVDLDR (A,B,W)
W = 2.*B/A
END
```

Calling Program References

```
CALL BLVDLDR (X(I),Y(I),W)
             .
             .
             .
CALL BLVDLDR (X(I)+H/2.,Y(I)
                              +C(1)/2.,W)
             .
             .
             .
CALL BLVDLDR (X(I)+H,Y(I)+C(3),Z)
```

2. *Subroutine Subprogram (Matrix Multiply)*

```
SUBROUTINE MATMULT
COMMON/BLK1/X(20,20),Y(20,20),
                    Z(20,20)
DO   10     I=1,20
DO   10     J=1,20
Z(I,J)=0.
DO   10    K=1,20
10 Z(I,J) = (I,J) + X(I,K)*Y(K,J)
RETURN
END
```

Calling Program Reference

```
COMMON/BLK1/A(20,20),B(20,20),
                   C(20,20)
             .
             .
             .
CALL MATMULT
             .
             .
             .
```

3. *Subroutine Subprogram*

```
  SUBROUTINE ISHTAR (Y,Z)
  COMMON/1/X(100)
  Z=0.
  DO 5 I=1,100
5 Z=Z+X(I)
  CALL Y
  RETURN
  END
```

Calling Program Reference

```
  COMMON/1/A(100)
  EXTERNAL PRNTIT
        .
        .
        .
  CALL ISHTAR (PRNTIT,SUM)
```

6.8 PROGRAM ARRANGEMENT

FORTRAN-66 assumes that all startements and comments appearing between a PROGRAM, SUBROUTINE or FUNCTION statement and an END statement belong to one program. A typical arrangement of a set of main program and subprograms follows.

```
/PROGRAM SOMTHING
{    .
{    .
{    .
\END

/SUBROUTINE S1
{    .
{    .
{    .
\END

/SUBROUTINE S2
{    .
{    .
{    .
\END
     .
     .
     .
/FUNCTION F1 (...)
{    .
{    .
{    .
\END

/FUNCTION F2 (...)
{    .
{    .
{    .
\END
```

6.9 RETURN AND END

A subprogram normally contains one or several RETURN statements that indicate the end of logic flow within the subprogram and return control to the calling program. The form is:

RETURN

In function references, control returns to the statement containing the function. In subroutine subprograms, control returns to the calling program.

The END statement marks the physical end of a program, subroutine subprogram or function subprogram. If the RETURN statement is omitted, END acts as a return to the calling program.

A RETURN statement in the main program causes an exit to the monitor.

6.10 ENTRY

This statement provides alternate entry points to a function or subroutine subprogram. Its form is:

ENTRY name

Name is an alphanumeric identifier, and may appear within the subprogram only in the ENTRY statement. Each entry identifier must appear in a separate ENTRY statement. The maximum number of entry points, including the subprogram name, is 20. The formal parameters, if any, appearing with the FUNCTION or SUBROUTINE statement do not appear with the ENTRY statement. ENTRY may appear anywhere within the subprogram except it must not appear within a DO; it can not be labeled.

In the calling program, the reference to the entry name is made just as though reference were being made to the FUNCTION or SUBROUTINE in which the ENTRY is imbedded. Rules 5 and 6 of 6.2 apply.

ENTRY names must agree in type with the function or subroutine name.

Examples:

```
       FUNCTION JOE (X,Y)
    10 JOE=X+Y
       RETURN
       ENTRY JAM
       IF(X.GT.Y)10,20
    20 JOE=X-Y
       END
```

This could be called from the main program as follows:

```
          .
          .
          .
    Z=A+B-JOE(3.*P,Q-1)
          .
          .
          .
    R=S+JAM(Q,2.*P)
```

6.11 VARIABLE DIMENSIONS IN SUBPROGRAMS

In many subprograms, especially those performing matrix manipulation, the programmer may wish to vary the dimension of the arrays each time the subprogram is called.

This is accomplished by specifying the array identifier and its dimensions as formal parameters in the FUNCTION or SUBROUTINE statement heading a subprogram. In the subroutine call from the calling program, the corresponding actual parameters specified are used by the called subprogram. The maximum dimension which any given array may assume is determined by a DIMENSION statement in the main program at compile time.

Rules:

1. The rules of 6.2, 6.5, and 6.7 apply.
2. The formal parameters representing the array dimensions must be simple integer variables. The array identifier must also be a formal parameter.
3. The actual parameters representing the array dimensions may be integer constants or integer variables.
4. If the total number of elements of a given array in the calling program is N, then the total number of elements of the corresponding array in the subprogram may not exceed N.

Examples:

1. Consider a simple matrix add routine written as a subroutine:

```
       SUBROUTINE MATADD(X,Y,Z,M,N)
       DIMENSION X (M,N),Y(M,N),Z(M,N)
       DO    10    I=1,M
       DO    10    J=1,N
    10 Z(I,J)=X(I,J)+Y(I,J)
       RETURN
       END
```

The arrays X,Y,Z and the variable dimensions M,N must all appear as formal parameters in the SUBROUTINE statement and also appear in the DIMENSION statement as shown. If the calling program contains the array allocation declaration

```
    DIMENSION A(10,10), B(10,10), C(10,10)
    DIMENSION D(4, 8), E(4, 8), F(4, 8)
    DIMENSION P(3, 12), Q(3, 12), R(3, 12)
```

the program may call the subroutine MATADD from several places within the main program, varying the array dimension within MATADD each time as follows:

```
    CALL MATADD (A,B,C,10,10)
                .
                .
                .
    CALL MATADD (D,E,F,4,8)
                .
                .
                .
    CALL MATADD (P,Q,R,3,12)
```

The compiler does not check to see if the limits of the array established by the DIMENSION statement in the main program are exceeded.

2.

$$Y = \left\{ \begin{array}{l} y_{11} \cdots y_{1n} \\ y_{21} \cdots y_{2n} \\ y_{31} \cdots y_{3n} \\ y_{41} \cdots y_{4n} \end{array} \right.$$

Its transposed Y′ is:

$$Y' = \left\{ \begin{array}{cccc} y_{11} & y_{21} & y_{31} & y_{41} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ y_{1n} & y_{2n} & y_{3n} & y_{4n} \end{array} \right.$$

The following 6600 FORTRAN program permits variation of N from call to call:

```
      SUBROUTINE MATRAN (Y, YPRIME, N)
      DIMENSION Y(4,N), YPRIME (N, 4)
      DO  7  I=1,N
      DO  7  J=1,4
 7    YPRIME (I,J)=Y(J,I)
      END
```

7. FORMAT SPECIFICATIONS

Data transmission between storage and an external unit requires the FORMAT statement and the I/O control statement (Chapter 9). The I/O statement specifies the input/output device and process—READ, WRITE, and so forth, and a list of data to be moved. The FORMAT statement specifies the manner in which the data are to be moved. In binary tape statements no FORMAT statement is used.

7.1 THE I/O LIST

The list portion of an I/O control statement indicates the data elements and the order, from left to right, of transmission. Elements may be simple variables or array names (subscripted or non-subscripted). They may be constants on output only. List elements are separated by commas, and the order must correspond to the order of the FORMAT specifications.

7.1.1 TRANSMISSION OF ARRAYS

Part or all of an array can be represented as a list item. Multi-dimensioned arrays may appear in the list, with values specified for the range of the subscripts in an implied DO loop.

The general forms are:

(a) transmission by columns

$(((A(I,J,K),I=m_1,m_2,m_3),J=n_1,n_2,n_3),$
$K=p_1,p_2,p_3)$

(b) transmission by rows

$(((A(I,J,K),K=p_1,p_2,p_3),J=n_1,n_2,n_3),$
$I=m_1,m_2,m_3)$

m_i,n_i,p_i	are any computable arithmetic expression. If m_3, n_3 or p_3 is omitted, it is construed as being 1.
I,J,K	are subscripts of A. If a subscript is omitted when transmitting an array, a DO statement is generated for the missing subscript using the entire range of values.

The I/O list may contain nested DO loops to any depth within the overall DO nest limit of 50 (including the nest depth in which the I/O statement resides).

Example:

DO loops nested 5 deep:

$(((((A(I,J,K),B(M),C(N),N=n_1,n_2,n_3),$
$M=m_1,m_2,m_3),K=k_1,k_2,k_3),$
$J=j_1,j_2,j_3),I=i_1,i_2,i_3)$

During execution, each subscript (index variable) is set to the initial index value: $I=i_1$, $J=j_1$, $K=k_1$, $M=m_1$, $N=n_1$.

The first index variable defined in the list is incremented first. Data named in the implied DO loops are transmitted in increments according to the third DO loop parameter until the second DO loop parameter is exceeded. If the third parameter is omitted, the increment value is 1. When the first index variable reaches the maximum value, it is reset; the next index variable to the right is incremented; and the process is repeated until the last index variable has been incremented.

An implied DO loop may also be used to transmit a simple variable more than one time. In (A,K =1,10), A will be transmitted 10 times.

Example:

As an element in an input/output list, the expression

$((A(I,J,K),I=m_1,m_2,m_3),J=n_1,n_2,n_3),$
$K=p_1,p_2,p_3)$

implies a nest of DO loops of the form

```
      DO    10    K=p1,p2,p3
      DO    10    J=n1,n2,n3
      DO    10    I=m1,m2,m3
Transmit A(I,J,K)
   10 CONTINUE
```

To transmit the elements of a 3 by 3 matrix by columns:

((A(I,J), I=1,3), J=1,3)

To transmit the elements of a 3 by 3 matrix by rows:

((A(I,J), J=1,3), I=1,3)

If a multi-dimensioned array name appears in a list without subscripts, the entire array is transmitted. For example, a multi-dimensioned, non-subscripted list element, SPECS, with an associated DIMENSION SPECS (7,5,3) statement is transmitted as though under control of the nested DO loops.

```
      DO    10    K=1,3
      DO    10    J=1,5
      DO     10     I=1,7
Transmit SPECS(I,J,K)
   10  CONTINUE
```

or as though under control of an implied DO loop,

```
      ..,( ( (SPECS(I,J,K), I=1,7), J=1,5),
      K=1,3),...
```

I/O Lists:

```
      ( (BUZ(K,2*L),K=1,5), L=1, 13,2)
      Q(3), Z(2,2), (TUP(3*I-4), I=2,10)
      (RAZ(K), K=1, LIM1, LIM2)
```

7.2 FORMAT STATEMENT

The BCD I/O control statements require a FORMAT statement which contains the specifications relating to the internal-external structure of the corresponding I/O list elements.

FORMAT $(\text{spec}_1, \ldots, k(\text{spec}_m, \ldots), \text{spec}_n, \ldots)$

Spec_i is a format specification and k is an optional repetition factor which must be an unsigned integer constant. The FORMAT statement is non-executable, and may appear anywhere in the program.

7.3 FORMAT SPECIFICATIONS

The data elements in I/O lists are converted from external to internal or from internal to external representation according to FORMAT conversion specifications. FORMAT specifications may also contain editing codes.

FORTRAN-66 conversion specifications

Ew.d	Single precision floating point with exponent
Fw.d	Single precision floating point without exponent
Dw.d	Double precision floating point with exponent
C(Zw.d,Zw.d)	Complex conversion; Z may be E or F conversion
Iw	Decimal integer conversion
Ow	Octal integer conversion
Aw	Alphanumeric conversion
Rw	Alphanumeric conversion
Lw	Logical conversion
nP	Scaling factor

FORTRAN-66 editing specifications

wX	Intra-line spacing
... wH	Heading and labeling
/	Begin new record
n/	Create n-1 blank records.

Both w and d are unsigned integer constants. w specifies the field width, the number of character positions in the record; and d specifies the number of digits to the right of the decimal within the field.

7.4 CONVERSION SPECIFICATIONS

7.4.1 EW.D OUTPUT

E conversion is used to convert floating point numbers in storage to the BCD character form for output. The field occupies w positions in the output record; the corresponding floating point number will appear right justified in the field as

| $\Delta\alpha.\alpha \ldots\ldots \alpha\text{E}\pm\text{eee} \qquad -307\leq\text{eee}\leq307$

$\alpha.\alpha \ldots\ldots \alpha$ are the most significant digits of the integer and fractional part and eee are the digits in the exponent. If d is zero or blank, the decimal point and digits to the right of the decimal do not appear as shown above. Field w must be wide enough to contain the significant digits, signs, decimal point, E, and the exponent. Generally, w is greater than or equal to d+7.

If the field is not wide enough to contain the output value, asterisks are inserted for the entire field. If the field is longer than the output value, the quantity is right justified with blanks in the excess positions to the left.

For P-scaling on output, see section 7.6.2.

Examples:

A contains – 67.32 or +67.32

```
      PRINT 10, A
10    FORMAT (E11.3)
      Result: -6.732E+001 or 6.732E+001
      PRINT 20, A
20    FORMAT (E14.3)
      Result: ^^^ -6.732E+001 or ^^^^ 6.732E+001
```

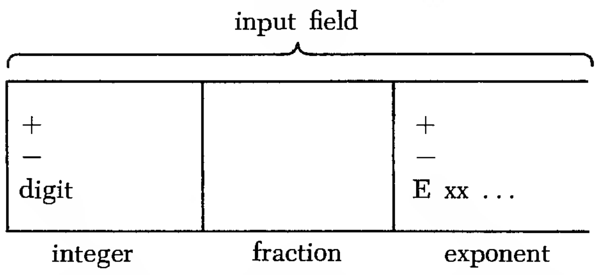
A contains – 67.32

```
      PRINT 30, A
30    FORMAT (E 10.3)
      Result: **********  Provision not made for
                          sign
      PRINT 40, A
40    FORMAT (E 11.4)
      Result: ***********  Same as above
```

7.4.2 EW.D INPUT

The E specification converts the number in the input field (specified by w) to a real and stores it in the appropriate location in memory.

Subfield structure of the input field:



The total number of characters in the input field is specified by w; this field is scanned from left to right; blanks within the field are interpreted as zeros.

The coefficient subfield, composed of an integer and/or a fraction, begins with a sign (+ or −) or a digit and contains a string of digits (a sequence of consecutive numbers). The fraction portion which begins with a decimal point contains a string of digits. The subfield is terminated by a D, E, a + or −, or the end of the input field.

An exponent subfield begins with a D, E, a + or a −. When it begins with E or D, the + or − appears between the E or D and the string of digits in the subfield. The value of this string of digits must be less than 310.

Permissible subfield combinations:

+1.6327E −04	integer fraction exponent
−32.7216	integer fraction
+328+5	integer exponent
.629E −1	fraction exponent
+136	integer only
.07628431	fraction only
E −06 (interpreted as zero)	exponent only

Rules:

1. In the Ew.d input specification, d acts as a negative power of ten scaling factor when the decimal point is not present. The internal representation of the input quantity will be:

   (integer subfield) $\times 10^{-d} \times 10^{(\text{exponent subfield})}$

   For example, if the specification is E7.8, the input quantity 3267+05 is converted and stored as: $3267 \times 10^{-8} \times 10^{5} = 3.267$.

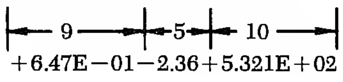
2. If E conversion is specified, but a decimal point occurs in the input constants, the decimal point will override d. The input quantity 3.67294+5 may be read by any specification but is always stored as 3.67294×10^{5}.

3. When d does not appear, it is assumed to be zero.

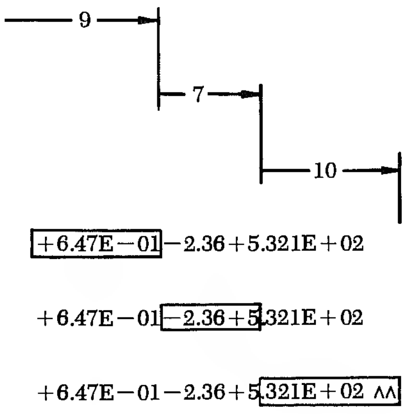
4. The field length specified by w in Ew.d must always be the same as the length of the input field containing the input number. When it is not, incorrect numbers may be read, converted and stored as shown below. The field w includes the significant digits, signs, decimal point, E, and exponent.

```
        READ 20,A,B,C
   20   FORMAT (E9.3,E7.2,E10.3)
```

The input quantities appear on a card in three contiguous field columns 1 through 24:



The second specification (E7.2) exceeds the physical field width of the second value by two characters. Reading proceeds as follows:



First +6.47E−01 is read, converted and placed in location A.

Next, $-2.36+5$ is read, converted and placed in location B. The number actually desired was -2.36, but the specification error (E7.2 instead of E5 2) caused the two extra characters to be read. The number read ($-2.36+5$) is a legitimate input representation under the definitions and restrictions.

Finally .321E+02 $_{\wedge\wedge}$ is read, converted and placed in location C. Here again, the input number is legitimate; and it is converted and stored, even though it is not the number desired.

The above example illustrates a situation where numbers are incorrectly read, converted, and stored, and yet there is no immediate indication that an error has occurred.

Examples:

Ew.d Input

Input Field	*Specification*	*Converted Value*	*Remarks*
+143.26E−03	E11.2	.14326	All subfields present
−12.437629E+1	E13.6	−124.37629	All subfields present
8936E+004	E9.10	.008936	No fraction subfield. Input number converted as 8936. $\times$ 10^{-10+4}
327.625	E7.3	327.625	No exponent subfield
4.376	E5	4.376	No d in specification
−.0003627+5	E11.7	−36.27	Integer subfield contains − only
−.0003627E5	E11.7	−36.27	Integer subfield contains − only
blanks	Ew.d	−0.	All subfields empty
1E1	E3.0	10.	No fraction subfield. Input number converted as 1. $\times$ 10^{1}
E+06	E10.6	0.	No integer or fraction subfield. Zero stored regardless of exponent field contents
1. E 1	E6.3	10.	Blanks are interpreted as zeros

7.4.3 FW.D OUTPUT

The field occupies w positions in the output record; the corresponding list element must be a floating point quantity, and it will appear as a decimal number; right justified in the field w, as:

$$\pm\ \delta \ldots \delta\ .\delta \ldots \delta$$

δ represents the most significant digits of the number. The number of places to the right of the decimal point is specified by d. If d is zero or omitted, the fractional digits do not appear. If the number is positive, the + sign is suppressed.

If the field is too short to accommodate the number, asterisks will appear in the output field.

If the field w is longer than required to accommodate the number, it is right justified with blanks occupying the excess field positions to the left.

Examples:

A contains +32.694

```
       PRINT 10,A
   10  FORMAT(F7.3)
       Result: ∧32.694

       PRINT 11,A
   11  FORMAT(F10.3)
       Result: ∧∧∧∧32.694
```

A contains −32.694

```
       PRINT 12,A
   12  FORMAT(F6.3)
       Result: ******      no provision for − sign
```

7.4.4 FW.D INPUT

This specification is a modification of Ew.d. The input field consists of an integer and a fraction subfield. An omitted subfield is assumed to be zero. All rules listed under Ew.d input apply. P-scaling is permissible.

Examples:

Fw.d Input

Input Field	*Specifi-cation*	*Converted Value*	*Remarks*
367.2593	F8.4	367.2593	Integer and fraction field
37925	F5.7	.0037925	No fraction subfield. Input number converted as 37925×10^{-7}
−4.7366	F7	−4.7366	No d in specification
.62543	F6.5	.62543	No integer subfield
.62543	F6.d	.62543	Decimal point overrides d of specification
+144.15E−03	F11.2	.14415	Exponents are legitimate in F input and may have P-scaling

7.4.5 DW.D OUTPUT

The field occupies w positions of the output record; the corresponding list element which must be a double precision quantity appears as a decimal number, right justified in the field w:

$\Delta\alpha.\alpha \ldots \alpha \, E \pm eee \qquad -307 \leqq eee \leqq 307$

D conversion corresponds to Ew.d output except that 29 is the maximum number of significant digits in the fraction.

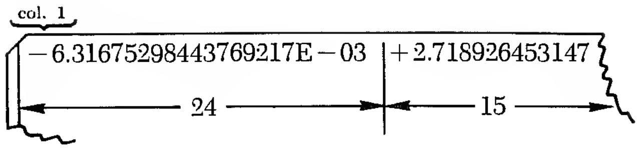
7.4.6 DW.D INPUT

D conversion corresponds to Ew.d input except that 29 is the maximum number of significant digits permitted in the combined integer-fraction field. P-scaling is not applicable. D is acceptable in place of E as the beginning of an exponent field.

Example:

```
      TYPE DOUBLE Z,Y
      READ1, Z,Y
   1  FORMAT (D24.17,D15)
```

Input card:



7.4.7 C ($Z_1W_1.D_1,Z_2W_2.D_2$) OUTPUT

Z is either E or F. The field occupies $w_1 + w_2$ positions in the output record, and the corresponding list element must be complex. $w_1 + w_2$ are two real values; w_1 represents the real part of the complex number and w_2 represents the imaginary part. The value may be one of the following forms:

$\Delta\delta.\delta \ldots \delta$ Exp.$\Delta\delta.\delta \ldots \delta$ Exp.	(Ew.d,Ew.d)
$\Delta\delta.\delta \ldots \delta$ Exp.$\Delta\delta \ldots \delta.\delta \ldots \delta$	(Ew.d,Fw.d)
$\Delta\delta \ldots \delta.\ \delta \ldots \delta\Delta\delta.\delta \ldots \delta$ Exp.	(Fw.d,Ew.d)
$\Delta\delta \ldots \delta.\delta \ldots \delta\Delta\delta \ldots \delta.\delta \ldots \delta$	(Fw.d,Fw.d)

Exp is $\pm e_1e_2e_3$.

The restrictions for Ew.d and Fw.d apply.

If spaces are desired between the two output numbers, the second specification should indicate a field (w_2) larger than required.

Example:

```
      TYPE COMPLEX A
      PRINT 10,A
  10  FORMAT (C(F7.2,F9.2) )
```

real part of A is 362.92
imaginary part of A is −46.73

Result: $\wedge$ 362.92 $\wedge\wedge\wedge$−46.73

7.4.8 C ($Z_1W_1.D_1,Z_2W_2.D_2$) INPUT

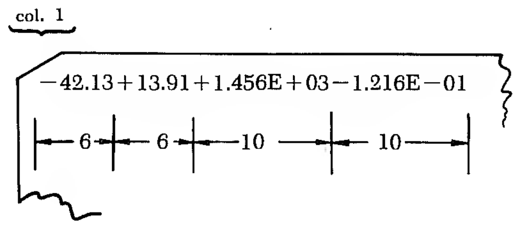
Z is either E or F and the input quantity occupies $w_1 + w_2$ character positions. The first w_1 characters are the representation of the real part of the complex number, and the remaining w_2 characters are the representation of the imaginary part of the complex number.

The restrictions for Ew.d and Fw.d apply.

Example:

```
     TYPE COMPLEX A,B
     READ 10,A,B
10   FORMAT (C(F6.2,F6.2), C(E10.3,E10.3) )
```

Input card:



7.4.9 IW OUTPUT

I specification is used to output decimal integer values. The output quantity occupies w output record positions; it will appear right justified in the field w, as:

Δδ . . . δ

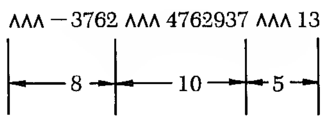
δ is the most significant decimal digits (maximum 18) of the integer. If the integer is positive, the + sign is suppressed.

If the field w is larger than required, the output quantity is right justified with blanks occupying excess positions to the left. If the field is too short, characters are discarded from the left.

Example:

```
     PRINT 10,I,J,K          I contains −3762
10   FORMAT (I8,I10,I5)      J contains +4762937
                             K contains +13
```

Result:



7.4.10 IW INPUT

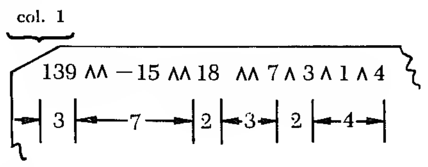
The field is w characters in length and the corresponding list element must be a decimal integer quantity.

The input field w which consists of an integer subfield may contain only the characters +, −, the digits 0 through 9, or blank. When a sign appears, it must precede the first digit in the field. Blanks are interpreted as zeros. The value is stored right justified in the specified variable.

Example:

```
     READ 10,I,J,K,L,M,N
10   FORMAT (I3,I7,I2,I3,I2,I4)
```

Input card:



In memory:

I contains	139
J	−1500
K	18
L	7
M	3
N	104

7.4.11 OW OUTPUT

O specification is used to output octal integer values. The output quantity occupies w output record positions, and it will appear right justified in the field as: δ δ . . . δ.

The δ are octal digits. If w is 20 or less, the rightmost w digits appear. If w is greater than 20, the number is right justified in the field with blanks to the left of the output quantity. A negative number is output in its machine form (one's complement).

7.4.12 OW INPUT

Octal integer values are converted under O specification. The field is w octal integer characters in length and the corresponding list element must be an integer quantity.

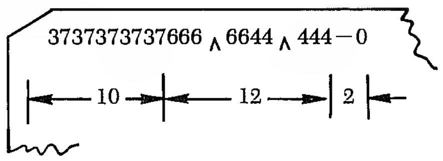
The input field w consists of an integer subfield only (maximum of 20 octal digits). The only characters that may appear in the field are +, or −, blank and 0 through 7. Only one sign is permitted;

it must precede the first digit in the field. Blanks are interpreted as zeros.

Example:

```
     TYPE INTEGER P,Q,R
     READ 10,P,Q,R
 10  FORMAT (O10,O12,O2)
```

Input Card:



In memory: P: 00000000003737373737

Q: 00000000660066440444

R: 77777777777777777777

A negative number is represented in complement form.

A negative octal number is represented internally in 20-digit seven's complement form obtained by subtracting each digit of an octal number from seven. For example, if −703 is an input quantity, its internal representation is 77777777777777777074.

That is,

```
  77777777777777777777
 −00000000000000000703
  77777777777777777074
```

7.4.13 AW OUTPUT

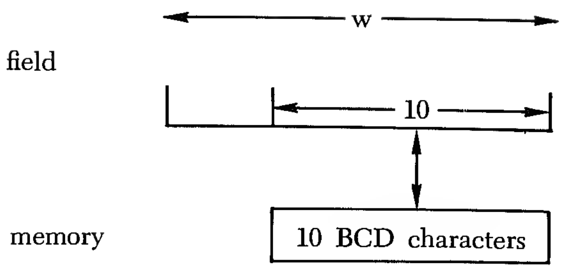
A conversion is used to output alphanumeric characters. If w is 10 or more, the output quantity appears right justified in the output field, blanks fill to left. If w is less than 10 the output quantity represents the leftmost w characters, left justified in the field.

7.4.14 AW INPUT

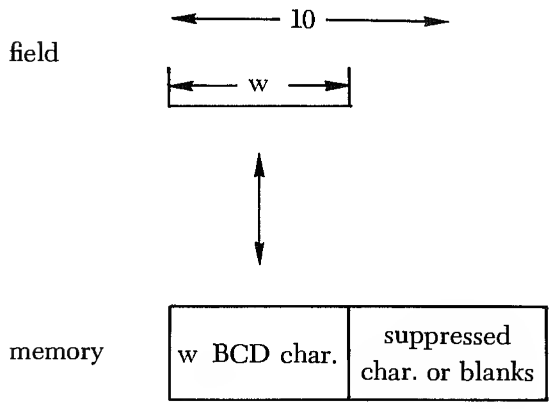
This specification will accept as list elements any set of six bit characters including blanks. The internal representation is BCD; the field width is w characters.

If w exceeds 10, the input quantity will be the rightmost characters. If w is 10 or less, the input quantity goes to the designated storage locations as a left justified BCD word, the remaining spaces are blank-filled.

w≧10 output
w>10 input



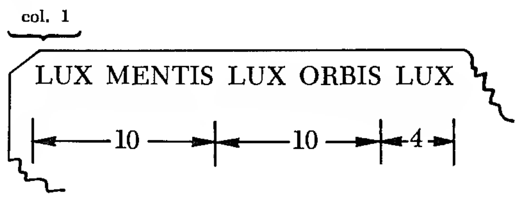
w<10 output
w≦10 input



Example: (Compare with next example)

```
     READ 10,Q,P,O
 10  FORMAT (A10,A10,A4)
```

Input card:



In memory: Q: LUXbMENTIS

P: bLUXbORBIS

O: bLUXbbbbbb

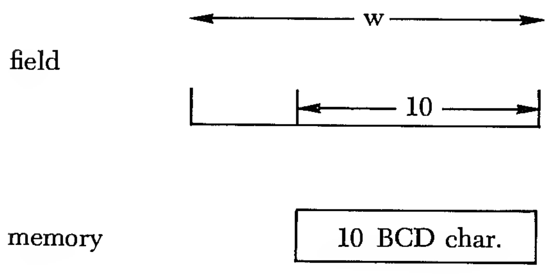
7.4.15 RW OUTPUT

This specification is the same as the Aw specification with the following exception. If w is less than 10 the output quantity represents the rightmost characters.

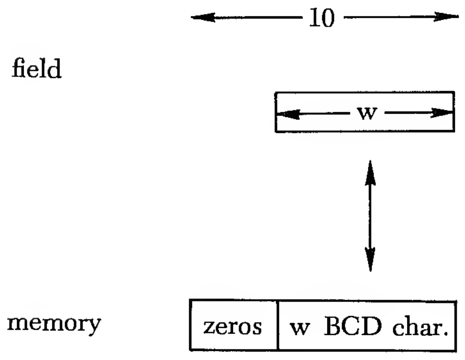
7.4.16 RW INPUT

If w is less than 10, the input quantity goes to the designated storage location as a right justified binary zero filled word.

$w \geq 10$ output
$w > 10$ input



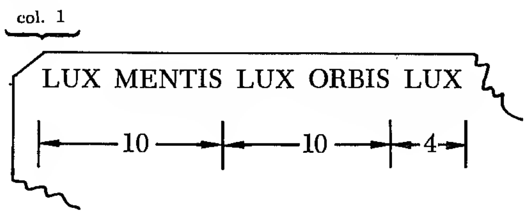
$w < 10$ output
$w \leq 10$ input



Example: (Compare with previous example)

```
   READ 10,Q,P,O
10 FORMAT (R10,R10,R4)
```

Input card:



In memory:		
	Q:	LUXbMENTIS
	P:	bLUXbORBIS
	O:	000000bLUX

7.4.17 LW OUTPUT

L specification is used to output logical values. The input/output field is w characters long and the corresponding list element must be a logical element.

If w is greater than 1, 1 or 0 is printed right justified in the field w with blank fill to the left.

Example:

```
  TYPE LOGICAL I,J,K,L        I contains 1
  PRINT 5,I,J,K,L             J contains 0
5 FORMAT (4L3)                K contains 1
                              L contains 1
```

Result: $_{\wedge\wedge}1_{\wedge\wedge}0_{\wedge\wedge}1_{\wedge\wedge}1$

7.4.18 LW INPUT

This specification will accept logical quantities as list elements. A zero or a blank in the field w is stored as zero. A one in the field w is stored as one. Only one such character (0 to 1) may appear in any input field. Any character other than 0, 1, or blank is incorrect.

7.5 EDITING SPECIFICATIONS

7.5.1 wX

This specification may be used to include w blanks in an output record or to skip w characters on input to permit spacing of input/output quantities.

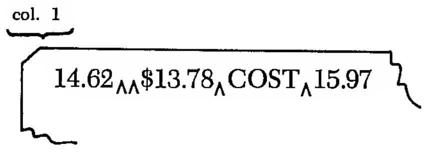
Examples:

```
   PRINT 10,A,B,C                 A contains 7
10 FORMAT                         B contains 13.6
   (I2,6X,F6.2,6X,E13.5)          C contains 1462.37
```

Result·$_\wedge$7←6→ $_\wedge$ 13.60←6→$_\wedge$1.46237E + 003

```
   READ 11,R,S,T
11 FORMAT(F5.2,3X, F5.2,6X,F5.2) or
   FORMAT (F5.2,3XF5.2,6XF5.2)
```

Input card: In memory: R = 14.62



S = 13.78

T = 15.97

In the specification list, the comma following X is optional.

7.5.2 WH OUTPUT

This specification provides for the output of any set of six-bit characters, including blanks, in the form of comments, titles, and headings. w is an unsigned integer specifying the number of characters to the right of the H that will be transmitted to the output record. H denotes a Hollerith field. The comma following the H specification is optional.

Examples:

Source program:

```
    PRINT 20
20  FORMAT(28H BLANKS COUNT IN AN H
    FIELD.)
```

produces output record:

```
∧BLANKS COUNT IN AN H FIELD.
```

Source program:

```
    PRINT 30,A                A contains 1.5
30  FORMAT(6H LMAX=,F5.2)
                              comma is optional
```

produces output record:

```
∧LMAX = ∧1.50
```

7.5.3 WH INPUT

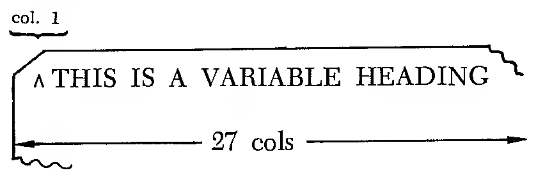
The H field may be used to read a new heading into an existing H field.

Example:

Source program:

```
    READ 10
10  FORMAT (27H ∧∧∧∧∧∧∧∧∧∧∧∧∧∧∧∧∧∧∧∧∧∧∧∧∧∧∧)
```

Input card:



∧THIS IS A VARIABLE HEADING

After the READ, the FORMAT statement labeled 10 will contain the alphanumeric information read from the input card; a subsequent reference to statement 10 in an output control statement would act as follows:

```
PRINT 10   produces the printer line:
           ∧THIS IS A VARIABLE HEADING
```

7.5.4 *N CHAR.* OUTPUT

This specification can be used as an alternate form of wH to output headings, titles, and comments. Any 6-bit character (except asterisk) between the asterisks will be output. The first asterisk denotes the beginning of the Hollerith field; the second asterisk terminates the field.

Examples:

1. Source program:

```
    PRINT 10
10  FORMAT (*∧SUBTOTALS*)
```

produces the output records: SUBTOTALS

2. If an asterisk accidentally occurs in the characters to be output, it will terminate the field and the characters following the asterisk will be interpreted as incorrect format specifications.

Source program:

```
    PRINT 10
10  FORMAT (*ABC*BE*)
```

produces the output record: ABC The conversion routine will attempt to interpret BE as a format specification.

7.5.5 *. . .* INPUT

This specification is used in place of wH to read a new heading into an existing Hollerith field. Characters are stored in the heading until an asterisk is encountered in the input field or until all the spaces in the format specification are filled. If the format specification contains n spaces and the mth charactem (m<n) in the input field is an asterisk, all characters to the left of the asterisk will be stored in the heading and the remaining character positions in the heading will be filled with blanks.

Examples:

1. Source program:

```
    READ 10
10  FORMAT (*∧∧∧∧∧∧∧∧∧∧∧∧∧∧∧∧∧∧∧∧*)
```

Input card: FORTRAN FOR THE 6600

A subsequent reference to statement 10 in an output control statement:

```
PRINT 10   produces:
    FORTRAN FOR THE 6600
```

2. Source program:

```
        READ 10
    10  FORMAT (*∧∧∧∧∧∧∧*)
```

Input card: ⌈HEAD*LINE

PRINT 10 produces: HEAD∧∧∧

7.5.6 NEW RECORD

The slash, /, which signals the end of a BCD record may occur anywhere in the specifications list. It need not be separated from the other list elements by commas; consecutive slashes may appear in a list. During output, it is used to skip lines, cards, or tape records. During input, it specifies that control passes to the next record or card. k+1 consecutive slashes produce k blank lines, or n/ produces n-1 blank lines during output and skips n-1 records or cards during input.

Examples:

```
     PRINT 10
 10  FORMAT (20X,7HHEADING///6X,
     5HINPUT,19X,6HOUTPUT)
Print-out:   HEADING                          line 1
                                              line 2
                                              line 3
     INPUT          OUTPUT                    line 4
```

Statement 10 in the example may also be written:

```
 10  FORMAT (20X, 7HHEADING3/6X,
     5HINPUT, 19X,6HOUTPUT)
```

Each line corresponds to a BCD record. The second and third records are null and produce the line spacing illustrated.

```
     Internally:
       A = -11.6              C =   46.327
       B =    .325            D = -14.261
     PRINT 11,A,B,C,D
 11  FORMAT (2E12.2/2F7.3)
     Result: ∧∧  -1.16E+001∧∧∧3.25E-001
             ∧46.327-14.261
     PRINT 11,A,B,C,D
 11  FORMAT (2E12.2/ /2F7.3)
     Result: ∧∧  -1.16E+001∧∧∧3.25E-001

             ∧46.327-14.261
     PRINT 15, (A(I),=1,9)
 15  FORMAT (8H RESULTS2/(3F8.2))
     Result:
     RESULTS

            3.62     -4.03     -9.78
           -6.33      7.12      3.49
            6.21     -6.74     -1.18
```

7.6 NP SCALE FACTOR

A scale factor may precede the F conversion and E conversion. The scale factor is: External number = Internal number $\times 10^{\text{scale factor}}$. The scale factor applies to Fw.d on both input and output and to Ew.d and Dw.d on output only. A scaled specification is written as:

$$nP \left\{ \begin{matrix} D \\ E \\ F \end{matrix} \right\} w.d$$

n is a signed integer constant. The nP specification may appear with complex conversion, C(Zw.d,Zw.d); each word is scaled separately according to Fw.d or Ew.d scaling.

7.6.1 FW.D SCALING

Input

The number in the input field is divided by 10^n and stored. For example, if the input quantity 314.1592 is read under the specification 2PF8.4, the internal number is $314.1592 \times 10^{-2} = 3.141592$.

Output

The number in the output field is the internal number multiplied by 10^n. In the output representation, the decimal point is fixed; the number moves to the left or right depending on whether the scale factor is plus or minus. For example, the internal number 3.1415926536 may be represented on output under scaled F specifications as follows:

Specification	*Output Representation*
F13.6	3.141593
1PF13.6	31.415927
3PF13.6	3141.592654
-1PF13.6	.314159

7.6.2 EW.D SCALING

Output

The scale factor has the effect of shifting the output number left n places while reducing the exponent by n. Using 3.1415926538 some output representations corresponding to scaled E-specifications are:

Specification	*Output Representation*
E20.2	3.14E+000
1PE20.2	31.42E-001
2PE20.2	314.16E-002
3PE20.2	3141.59E-003
4PE20.2	31415.93E-004
5PE20.2	314159.27E-005
-1PE20.2	0.31E+001

7.6.3 SCALING RESTRICTIONS

The scale factor is assumed to be zero if no other value has been given; however, once a value has been given, it will hold for all D,E and F specifications following the scale factor within the same FORMAT statement. To nullify this effect in subsequent D,E and F specifications, a zero scale factor, 0 P, must precede a D,E, or F specification. Scale factors on D or E input specifications are ignored.

The scaling specification nP may appear independently of an E or F specification, but it will hold for all E and F specifications that follow within the same FORMAT statement unless changed by another nP.

```
(3P, 3I9, F10.2) same as
(3I9, 3PF10.2)
```

7.7 REPEATED FORMAT SPECIFICATIONS

Any FORMAT specification may be repeated by using an unsigned integer constant repetition factor, k, as follows: k(spec), spec is any conversion specification except nP. For example, to print two quantities K,L:

```
    PRINT 10 K,L
10  FORMAT (I2,I2)
```

Specifications for K,L are identical; the FORMAT statement may be: 10 FORMAT (2I2).

When a group of FORMAT specifications repeats itself, as in: FORMAT (E15.3,F6.1,I4,I4,E15.3,F6.1, I4,I4), the use of k produces: FORMAT (2(E15.3, F6.1,2I4)).

The parenthetical grouping of FORMAT specifications is called a repeated group. Repeated groups may be nested to 10 levels.

FORMAT (n_1(---n_2(-----n_{10}))))))))))

Therefore, FORMAT statements like the following example are legal.

```
FORMAT (1H1,5(25X,13(5X,F6.2) ) )
```

7.7.1 UNLIMITED GROUPS

FORMAT specifications may be repeated without the use of a repetition factor. The innermost parenthetical group that has no repetition factor is unlimited and will be used repeatedly until the I/O list is exhausted. Parentheses are the controlling factors in repetition. The right parenthesis of an unlimited group is equivalent to a slash. Specifications to the right of an unlimited group can never be reached.

The following are format specifications for output data:

```
(E16.3,F20.7,(2I4,2(I3,F7.1) ),F8.2)
```

Print fields according to E16.3 and F20.7. Since 2(I3,F7.1) is a repeated parenthetical group, print fields according to (2I4,2)I3,F7.1)), which does not have repetition operator, until the list elements are exhausted. F8.2 will never be reached.

7.8 VARIABLE FORMAT

FORMAT lists may be specified at the time of execution. The specification list including left and right parentheses, but not the statement number or the word FORMAT, is read under A conversion or in a DATA statement and stored in an integer array. The name of the array containing the specifications may be used in place of the FORMAT statement number in the associated input/output operation. The array name that appears with or without subscript specifies the location of the first word of the FORMAT information.

Examples:

1. Assume the following FORMAT specifications:

   ```
   (E12.2,F8.2,I7,2E20.3,F9.3,I4)
   ```

   This information could be punched in an input card and read by a program such as:

   ```
     DIMENSION IVAR(4)
     READ 1, (IVAR(I),I = 1,4)
   1 FORMAT(3A8,A6)
   ```

   The elements of the input card will be placed in storage as follows:

   ```
   IVAR    :   (E12.2,F
   IVAR+1  :   8.2,I7,2
   IVAR+2  :   E20.3,F9
   IVAR+3  :   .3,I4)bb
   ```

A subsequent output statement in the same program could refer to these FORMAT specifications as:

```
     PRINT IVAR(1),A,B,I,C,D,E,J
or
     PRINT IVAR,A,B,I,C,D,E,J
```

This would produce exactly the same result as the program:

```
     PRINT 10,A,B,I,C,D,E,J
  10 FORMAT (E12.2,F8.2,I7,2E20.3,F9.3,I4)
```

2. DIMENSION LAIS(4)
```
   DATA (LAIS=8H(E12.2,F8H8.2,2I7),
                 8H(F8.2,E1,8H2.2,2I7) )
```

Output statements:

When I=1

```
          PRINT LAIS (I),A,B,I,J
or
          PRINT LAIS,A,B,I,J
```

This is also the same as:

```
          PRINT 1,A,B,I,J
1         FORMAT  (E12.2,F8.2,2I7)
```

When I=3

```
          PRINT LAIS (I),C,D,I,J
```

This is the same as:

```
          PRINT 2,C,D,I,J
2         FORMAT (F8.2,E12.2,2I7)
```

8. INPUT-OUTPUT STATEMENTS

Input-output statements transfer information between the memory unit and one of the following external devices:

An 80 column card reader

An 80 column card punch

A 136 column printer

A magnetic tape unit

The external unit, i, may be specified by an integer variable or a constant, $1 \leq i \leq 49$. Logical values are established for physical units by SIPROS.

The FORMAT statement number for BCD data transmission is represented by n; and may be the statement number, a variable identifier or a formal parameter. Binary data transmission does not require a FORMAT statement.

The input-output list is specified by L. Binary information is transmitted with odd parity checking bits. BCD information is transmitted with even parity checking bits.

8.1 WRITE STATEMENTS

PRINT n,L

Transfers information from the memory locations given by the list (L) identifiers to the standard output medium. This information is transferred as line printer images in accordance with the FORMAT specification n.

PUNCH n,L

transfers information from the memory locations given by the list (L) identifiers to the standard punch medium. This information is transferred as card images in accordance with the FORMAT specification n.

WRITE (i,n)L

and WRITE OUTPUT TAPE i,n,L are equivalent forms which transfer information from memory to a specified unit (i) using the number of list (L) elements and the FORMAT specification (n) to determine the number of records that will be written on that unit. Each logical record is one physical record containing up to 136 characters; the information is recorded in even parity (BCD mode).

WRITE(i)L

and WRITE TAPE i,L are equivalent forms which transfer information from memory to a specified unit (i) using the number of list elements (L) to determine the number of logical records that will be written on that unit. If there is only one physical record in the logical record, the first word contains the integer 1.

If there are n physical records in the logical record, the first word of the first n − 1 physical records contain zero; the first word of the *nth* physical record contains the integer n. This first word indicates how many physical records exist in a logical record, and it is used in the BACKSPACE Statement. The information is recorded in odd parity (binary mode). If the LIST is omitted, the WRITE(i) statement acts as a do-nothing.

Examples:

```
      WRITE OUTPUT TAPE 10,20,A,B,C
 20   FORMAT (3F10.6)

      WRITE (10,20) Q1,Q2,Q3

      DIMENSION A(260), B(4)
        .
        .
        .
      WRITE (10) A,B

      DO 5 I=1, 10
  5   WRITE TAPE 6,AMAX(I), (M(I,J),J=1,5)

      WRITE OUTPUT TAPE 4,21
 21   FORMAT (33H    THIS STATEMENT HAS
                     NO DATA LIST.)
```

8.2 READ STATEMENTS

READ n,L

reads one or more card images, converting the information from left to right, in accordance with the FORMAT specification (n), and stores the converted data in the memory locations named by the list (L) identifiers. The images read may come from 80-column Hollerith cards, or from magnetic tapes, prepared off-line, containing 80-character records in BCD mode.

READ(i,n)L

and READ INPUT TAPE i,n,L are equivalent forms which transfer information from a specified logical unit (i) to memory locations named by the list (L) identifiers. The number of list elements and the FORMAT specifications must conform to the record structure on the logical unit (up to 136 characters in the BCD mode).

READ(i)L

and READ TAPE i,L are equivalent forms which transfer information from a specified logical unit (i) to memory. The number of words in the list must be equal to or less than the number of words in the record on the logical unit. A record read by READ (i)L should be the result of a binary mode WRITE statement. If the statement occurs without a list, READ(i), the tape on unit i moves forward one logical record.

Examples:

```
       READ 10,A,B,C
10     FORMAT (3F10.4)
       READ (2,13) (Z(K),K=1,8)
13     FORMAT (F10.4)
       READ INPUT TAPE 6,20, Q1,Q2,Q3,Q4
20     FORMAT (4E11.3)
       READ (6) ( (A(I,J),I=1,100),J=1,50)
       READ TAPE 6, ( (A(I,J),I=1,100),J=1,50)
       READ (5)   (skip one logical record on unit 5)
```

8.3 BUFFER STATEMENTS

There are three primary differences between the buffer I/O control statements and the read/write I/O control statements.

1. The mode of transmission (BCD or binary) is tacitly implied by the form of the read/write control statement. In a buffer control statement, parity must be specified by a parity indicator.
2. The read/write control statements are associated with a list, and in BCD transmission with a FORMAT statement. The buffer control statements are not.
3. A buffer control statement initiates data transmission, and then returns control to the program, permitting the program to perform other tasks while data transmission is in progress.

In the descriptions that follow, these definitions hold:

i	logical unit number: integer constant or variable.
p	parity indicator: 0 for even parity (BCD); 1 for odd parity (binary).
A	variable identifier: first word of data block to be transmitted.
B	variable identifier: last word of data block to be transmitted. The location represented by A must be equal to or less than the location represented by B.

BUFFER IN (i,p) (A,B)

transmits information from unit i in mode p to memory locations A through B. If a magnetic tape containing BCD records written by WRITE(i,n) is used by BUFFER IN, only one physical record will be read. If a magnetic tape written by WRITE(i) is read by BUFFER IN, only one physical record is read.

BUFFER OUT (i,p) (A,B)

transmits information from memory locations A through B, to unit i in mode p. The physical record written contains B−A+1 words, or a maximum of 512 words.

8.4 TAPE HANDLING STATEMENTS

The logical unit number, i, may be an integer variable or constant.

REWIND i

rewinds the magnetic tape mounted on unit i. If tape is already rewound, the statement acts as a do-nothing.

BACKSPACE i

backspaces the magnetic tape mounted on unit i one logical record. (A logical record is a physical record; except for tapes written by a WRITE(i)L statement). If tape is at load point (rewound) this statement acts as a do-nothing.

END FILE i

writes an end-of-file on the magnetic tape mounted on unit i.

The tape handling statements cannot be used on the standard input, standard output, or standard punch media.

8.5 STATUS CHECKING STATEMENTS

IF(EOF,i)n_1,n_2

checks the previous read (write) operation to determine if an end-of-file (end-of-tape) has been encountered on unit i. If it has, control is transferred to statement n_1; if not, control is transferred to statement n_2.

IF(IOCHECK,i)n_1,n_2

checks the previous read (write) operation to determine if a parity error has occurred on unit i. If it has, control is transferred to statement n_1; if not, control is transferred to statement n_2.

IF(UNIT,i)n_1,n_2,n_3,n_4

is the status checking statement used with buffer control. It should always appear before the first statement that uses any of the variables transferred in the buffer mode to avoid loss of information. The n_i are statement numbers and the following criteria apply:

Check the status of the last buffering operation on unit i and transfer control to statement

n_1 if buffer operation is not complete

n_2 if buffer operation is complete and NO error occurred

n_3 if buffer operation is complete and an EOF or EOT occurred

n_4 if buffer operation is complete and parity or buffer length errors occurred.

A buffer length error occurs on read only to indicate that a record of length k words has been read into a buffer area where word length is greater than k.

Example:

Program		*Remarks*
	J = I	Set flag = 1
	BUFFER IN (10,0) (A,Z)	Initiate buffered read in even (BCD) parity.
4	IF(UNIT,10)5,6,7,8	Check status of buffered transfer.
5	GO TO (50,4),J	Not finished. Do calculations at 50.
50	{Some computation not involving information in locations A–Z}	
	J = J + I GO TO 4	Calculations complete; increase flag by I. Go to 4.
7 70	PRINT 70 FORMAT(I2H EOF UNIT 10) GO TO 200	End-of-file error.
8	PRINT 80	
80	FORMAT(35H PARITY OR BUF LENGTH ERROR UNIT I0)	
200	REWIND I0	Rewind tape and stop.
	STOP	Stop.
6	CONTINUE . . .	Buffer transmission complete. Continue program.

8.6 ENCODE/DECODE STATEMENTS

The ENCODE/DECODE statements are comparable to the WRITE/READ statements, the essential difference being that no peripheral equipment is used in the data transfer. Information is transferred under FORMAT specifications from one area of memory to another.

In the following descriptions, n and L have the same meanings as were defined for input-output statements.

c is an integer constant or integer variable specifying the record length (characters).

V is a variable identifier or an array identifier.

ENCODE (c,n,V)L

The information of the list variables (L) is converted according to the n FORMAT statement and stored in the locations identified by V.

DECODE (c,n,V)L

The information in the locations identified by V is converted according to the n FORMAT statement and stored in the list variables (L).

Example:

1. The following is one method of packing the partial contents of two words into one word. Information is stored in core as follows:

```
      LOC(1)SSSSSxxxxx
        .
        .
        .
      LOC(6)xxxxxααααα          10 BCD ch/wd.
```

To form SSSSSααααα in storage location NAME:

```
      DECODE(10,1,LOC(6) )TEMP
   1  FORMAT(5X,A5)
      ENCODE(10,2,NAME) LOC(1),TEMP
   2  FORMAT(2A5)
```

The DECODE statement places the last 5 BCD characters of LOC(6) into the first 5 characters of TEMP. The ENCODE statement packs the first 5 characters of LOC(1) and TEMP into NAME.

A more straightforward way of accomplishing this is with masking statements. Using the R specification, the program may be shortened to:

```
      ENCODE (10,1,NAME) LOC(1),LOC(6)
   1  FORMAT (A5,R5)
```

2. DECODE may be used to calculate a field definition in a FORMAT specification at object time. Assume that in the statement FORMAT (2A10, Im) the programmer wishes to specify m at some point in the program, subject to the restriction $2 \leq m \leq 9$. The following program permits m to vary.

```
      IF(M .LT. 10 .AND. M .GT. 1)1,2
   1  ENCODE (10,100,SPECMAT) M
 100  FORMAT (7H(2A10,I,I1,1H) )
        .
        .
        .
      PRINT SPECMAT,A,B,J
```

M is tested to insure it is within limits. If not, control goes to statement 2 which could be an error routine. If M is within limits, ENCODE packs the integer value of M with the characters: (2A10,I). This packed FORMAT is stored in SPECMAT. SPECMAT contains (2A10,Im).

The print statement will print A and B under specification A10, and the quantity J under specification I2 ,or I3 or . . . or I9 according to the value of m.

APPENDIX

A. CODING PROCEDURES

CODING FORM

FORTRAN-66 forms contain 80 columns in which the characters of the language are written, one character per column.

STATEMENTS

The statements of FORTRAN-66 are written in columns 7 through 72. Statements longer than 66 columns may be carried to the next line by using a continuation designator. More than one statement may be written on a line. Blanks may be used freely in FORTRAN statements to provide readibility. Blanks are significant, however, in H fields.

STATEMENT SEPARATOR $

The special character $ may be used to write more than one statement on a line. Statements so written may also use the continuation feature. A $ symbol may not be used as a statement separator with FORMAT statements or continuations of FORMAT statements.

These statements are equivalent:

```
      I=10                  I=10 $ JLIM=1 $
      JLIM=1                K=K+1 $ GO TO 10
      K=K+1
      GO TO 10
```

Also:

```
      DO 1 I=1, 10          DO 1 I=1, 10 $ A(I)
      A(I)=B(I)+C(I)        =B(I)+C(I)
    1 CONTINUE            1 CONTINUE $ I=3
      I=3
```

COMMENT CARD

Comment information is designated by a C in column 1 of a statement. Comment information will appear in the source program, but it is not translated into object code. Columns 2 through 80 may be used. Continuation is not permitted; that is, each line of comments must be preceded by the C designator in column 1.

All comment cards belonging to a specific program, or subprogram, should appear between the PROGRAM, SUBROUTINE, or FUNCTION statement and the END statement.

STATEMENT IDENTIFIERS

Any statement may have an identifier but only statements referred to elsewhere in the program require identifiers. A statement identifier is a string of 1 to 5 digits occupying any column positions 1 through 5.

If I is an identifier, $1 \leq I \leq 99999$; leading zeros are ignored; 1, 01, 001, 0001 are equivalent forms. Zero is not a statement identifier. In any given program or subprogram each statement identifier must be unique. If the statement identifier is followed by a character other than blank in column 6, the statement identifier is ignored.

Statement identifiers of Declarative statements (excepting FORMAT) are ignored by the compiler, except for diagnostic purposes.

CONTINUATION

The first line of every statement must have a blank in column 6. If statements occupy more than one line, all subsequent lines must have a FORTRAN character other than blank or zero in column 6. A FORTRAN-66 statement may have up to 660 operators, delimiters (commas or parentheses) and identifiers within it. The compiler will accept up to nine continuation cards.

IDENTIFICATION FIELD

Columns 73 through 80 are always ignored in the translation process. These columns, therefore, may be used for identification when the program is to be punched on cards. Usually these columns contain sequencing information provided by the programmer.

PUNCHED CARDS

Each line of the coding form corresponds to one 80-column card; the terms line and card are often used interchangeably. Source programs and data can be read into the computer from cards; a relocatable binary deck or data, can be punched directly onto cards.

Blank cards appearing within the input card deck are ignored.

When cards are being used for data input, all 80 columns may be used.

CARRIAGE CONTROL

When printing on-line, the first character of a record transmitted by a PRINT statement is a carriage control character for spacing on the printer. The carriage control characters are:

Character	*Action*
Blank or any character other than the following	Single space after printing
0	Double space before printing
1	Eject page before printing
+	Suppress spacing after printing, causing two successive records to be printed on the same line.

The characters, 0 1 +, are not printed; any other causes single spacing and is not printed.

When printing off-line, the printer control is determined by a printer routine for a particular installation.

B. CENTRAL PROCESSOR OPERATION CODES

Octal Opcode	Mnemonic	Address	Comments
			. **BRANCH UNIT**
00	PS		. Program stop
01	RJ	K	. Return jump to K
02	JP	Bi + K	. Jump to Bi + K
030	ZR	Xj K	. Jump to K if Xj = 0
031	NZ	Xj K	. Jump to K if Xj ≠ 0
032	PL	Xj K	. Jump to K if Xj = plus (positive)
033	NG	Xj K	. Jump to K if Xj = negative
034	IR	Xj K	. Jump to K if Xj is in range
035	OR	Xj K	. Jump to K if Xj is out of range
036	DF	Xj K	. Jump to K if Xj is definite
037	ID	Xj K	. Jump to K if Xj is indefinite
04	EQ	Bi Bj K	. Jump to K if Bi = Bj
04	ZR	Bi K	. Jump to K if Bi = B0
05	NE	Bi Bj K	. Jump to K if Bi ≠ Bj
05	NZ	Bi K	. Jump to K if Bi ≠ B0
06	GE	Bi Bj K	. Jump to K if Bi ≥ Bj
06	PL	Bi K	. Jump to K if Bi ≥ B0
07	LT	Bi Bj K	. Jump to K if Bi < Bj
07	NG	Bi K	. Jump to K if Bi < B0
			. **BOOLEAN UNIT**
10	BXi	Xj	. Transmit Xj to Xi
11	BXi	Xj*Xk	. Logical Product of Xj & Xk to Xi
12	BXi	Xj + Xk	. Logical sum of Xj & Xk to Xi
13	BXi	Xj − Xk	. Logical difference of Xj & Xk to Xi
14	BXi	− Xk	. Transmit the comp. of Xk to Xi
15	BXi	− Xk*Xj	. Logical product of Xj & Xk comp. to Xi
16	BXi	− Xk + Xj	. Logical sum of Xj & Xk comp. to Xi
17	BXi	− Xk − Xj	. Logical difference of Xj & Xk comp. to Xi
			. **SHIFT UNIT**
20	LXi	jk	. Left shift Xi, jk places
21	AXi	jk	. Arithmetic right shift Xi, jk places
22	LXi	Bj Xk	. Left shift Xk nominally Bj places to Xi
23	AXi	Bj Xk	. Arithmetic right shift Xk nominally Bj places to Xi
24	NXi	Bj Xk	. Normalize Xk in Xi and Bj
25	ZXi	Bj Xk	. Round and normalize Xk in Xi and Bj
26	UXi	Bj Xk	. Unpack Xk to Xi and Bj
27	PXi	Bj Xk	. Pack Xi from Xk and Bj
43	MXi	jk	. Form mask in Xi, jk bits

Octal Opcode	Mnemonic	Address	Comments
			. **ADD UNIT**
30	FXi	Xj + Xk	. Floating sum of Xj and Xk to Xi
31	FXi	Xj − Xk	. Floating difference Xj and Xk to Xi
32	DXi	Xj + Xk	. Floating DP sum of Xj and Xk to Xi
33	DXi	Xj − Xk	. Floating DP difference of Xj and Xk to Xi
34	RXi	Xj + Xk	. Round floating sum of Xj and Xk to Xi
35	RXi	Xj − Xk	. Round floating difference of Xj and Xk to Xi
			. **LONG ADD UNIT**
36	IXi	Xj + Xk	. Integer sum of Xj and Xk to Xi
37	IXi	Xj − Xk	. Integer difference of Xj and Xk to Xi
			. **MULTIPLY UNIT**
40	FXi	Xj * Xk	. Floating product of Xj and Xk to Xi
41	RXi	Xj * Xk	. Round floating product of Xj & Xk to Xi
42	DXi	Xj * Xk	. Floating DP product of Xj & Xk to Xi
			. **DIVIDE UNIT**
44	FXi	Xj / Xk	. Floating divide Xj by Xk to Xi
45	RXi	Xj / Xk	. Round floating divide Xj by Xk to Xi
46	NO		. No operation
47	CXi	Xk	. Count the number of 1's in Xk to Xi
			. **INCREMENT UNIT**
50	SAi	Aj + K	. Set Ai to Aj + K
50	SAi	Aj − K	. Set Ai to Aj + comp. of K
51	SAi	Bj + K	. Set Ai to Bj + K
51	SAi	Bj − K	. Set Ai to Bj + comp. of K
52	SAi	Xj + K	. Set Ai to Xj + K
52	SAi	Xj − K	. Set Ai to Xj + comp. of K
53	SAi	Xj + Bk	. Set Ai to Xj + Bk
54	SAi	Aj + Bk	. Set Ai to Aj + Bk
55	SAi	Aj − Bk	. Set Ai to Aj − Bk
56	SAi	Bj + Bk	. Set Ai to Bj + Bk
57	SAi	Bj − Bk	. Set Ai to Bj − Bk
60	SBi	Aj + K	. Set Bi to Aj + K
60	SBi	Aj − K	. Set Bi to Aj + comp. of K
61	SBi	Bj + K	. Set Bi to Bj + K
61	SBi	Bj − K	. Set Bi to Bj + comp. of K
62	SBi	Xj + K	. Set Bi to Xj + K
62	SBi	Xj − K	. Set Bi to Xj + comp. of K
63	SBi	Xj + Bk	. Set Bi to Xj + Bk
64	SBi	Aj + Bk	. Set Bi to Aj + Bk
65	SBi	Aj − Bk	. Set Bi to Aj − Bk
66	SBi	Bj + Bk	. Set Bi to Bj + Bk
67	SBi	Bj − Bk	. Set Bi to Bj − Bk

Octal Opcode	Mnemonic	Address	Comments
70	SXi	AJ + K	· Set Xi to Aj + K
70	SXi	Aj − K	· Set Xi to Aj + comp. of K
71	SXi	Bj + K	· Set Xi to Bj + K
71	SXi	Bj − K	· Set Xi to Bj + comp. of K
72	SXi	Xj + K	· Set Xi to Xj + K
72	SXi	Xj − K	· Set Xi to Xj + comp. of K
73	SXi	Xj + Bk	· Set Xi to Xj + Bk
74	SXi	Aj + Bk	· Set Xi to Aj + Bk
75	SXi	Aj − Bk	· Set Xi to Aj − Bk
76	SXi	Bj + Bk	· Set Xi to Bj + Bk
77	SXi	Bj − Bk	· Set Xi to Bj − Bk

C. STATEMENTS OF FORTRAN-66

Subprogram Statements

Entry Points	PROGRAM name	N*
	SUBROUTINE name	N
	SUBROUTINE name (p_1,p_2, . . .)	N
	FUNCTION name (p_1,p_2, . . .)	N
	REAL FUNCTION name (p_1p_2, . . .)	N
	INTEGER FUNCTION name (p_1,p_2, . . .)	N
	DOUBLE PRECISION FUNCTION name (p_1,p_2 . . .)	N
	COMPLEX FUNCTION name (p_1,p_2, . . .)	N
	LOGICAL FUNCTION name (p_1,p_2, . . .)	N
	ENTRY name	N
Inter-subroutine	EXTERNAL $name_1$,$name_2$, . . .	N
Transfer Statements	CALL name	E
	CALL name (p_1, . . . ,p_n)	E
	RETURN	E

Data Declaration and Storage Allocation

Type Declaration	TYPE COMPLEX List	N
	TYPE DOUBLE List	N
	TYPE REAL List	N
	TYPE INTEGER List	N
	TYPE LOGICAL List	N
	COMPLEX List	N
	DOUBLE PRECISION List	N
	REAL List	N
	INTEGER LIST	N
	LOGICAL List	N
Storage Allocations	DIMENSION V_1,V_2, . . .,V_n	N
	COMMON/I_i/List	N
	EQUIVALENCE (A,B, . . .), (A1,B1, . . .) . . .	N
	DATA (I_1 = List), (I_2 = List), . . .	N
	BANK, (b_1), $name_1$, . . . , (b_2), $name_2$, . . .	N

Arithmetic Statement Function

	Function (p_1,p_2, . . . p_n) = Expression	E

Symbol Manipulation, Control and I/O

Replacement	A = E Arithmetic	E
	L = E Logical/Relational	E
	M = E Masking	E
	A_m= . . . = A_1 = E Multiple	E

* N = Non-executable E = Executable

Intra-program Transfers	GO TO n	E
	GO TO m, (n, . . . n_m)	E
	GO TO (n_1, . . . ,n_m)i	E
	GO TO (n_1, . . . ,n_m),i	E
	IF (A) n_1,n_2,n_3	E
	IF (L) n_1n_2	E
	IF (L)s	E
	IF (SENSE LIGHT i) n_1,n_2	E
	IF (SENSE SWITCH i) n_1,n_2	N
	IF DIVIDE $\left\{\begin{matrix}\text{FAULT}\\ \text{CHECK}\end{matrix}\right\}$ n_1,n_2	E
	IF EXPONENT FAULT n_1,n_2	E
	IF OVERFLOW FAULT n_1,n_2	E
	IF (EOF,i) n_1,n_2	E
	IF (IOCHECK), i) n_1,n_2	E
	IF (UNIT,i) n_1,n_2,n_3,n_4	E
	IF (UNIT,i) n_1,n_2,n_3	E
	IF (UNIT,i) n_1,n_2	E
Loop Control	DO $n_i = m_1,m_2,m_3$	E
Miscellaneous Program Controls		
	ASSIGN s TO m	E
	SENSE LIGHT i	E
	CONTINUE	E
	PAUSE; PAUSE n	E
	STOP; STOP n	E
I/O Format	FORMAT ($spec_1$,$spec_2$, . . .)	N
I/O Control Statements	READ n,L	E
	PRINT n,L	E
	PUNCH n,L	E
	READ (i,n) L READ INPUT TAPE i,n,L	E
	WRITE (i,n) L WRITE OUTPUT TAPE i,n,L	E
	READ (i) L READ TAPE i,L	E
	WRITE (i) L WRITE TAPE i,L	E
	BUFFER IN (i,p) (A,B) BUFFER OUT (i,p) (A,B)	E
I/O Tape Handling	END FILE i	E
	REWIND i	E
	BACKSPACE i	
Internal Data Manipulation	ENCODE (c,n,V) L	E
	DECODE (c,n,V) L	E
Program and Subroutine Termination		
	END	N/E

D. LIBRARY FUNCTIONS

Form	*Definition*	*Actual Parameter Type*	*Mode of Result*
ABSF(X)	Absolute Value	Real	Real
XABSF(i)		Integer	Integer
INTF(X)	Truncation, integer	Real	Real
XINTF(X)		Real	Integer
MODF(X_1,X_2)	X_1 modulo X_2	Real	Real
XMODF (i_1,i_2)	i_1 modulo i_2	Integer	Integer
MAX0F(i_1,i_2, . . .)	Determine maximum argument	Integer	Real
MAX1F(X_1,X_2, . . .)		Real	Real
XMAX0F(i_1,i_2, . . .)		Integer	Integer
XMAX1F(X_1,X_2, . . .)		Real	Integer
MIN0F(i_1,i_2, . . .)	Determine minimum argument	Integer	Real
MIN1F(X_1,X_2, . . .)		Real	Real
XMIN0F($i_1$$i_2$, . . .)		Integer	Integer
XMIN1F(X_1,X_2, . . .)		Real	Integer
SINF(X)	Sine X radians	Real	Real
COSF(X)	Cosine X radians	Real	Real
TANF(X)	Tangent X radians	Real	Real
ASINF(X)	Arcsine X radians	Real	Real
ACOSF(X)	Arccos X radians	Real	Real
ATANF(X)	Arctangent X radians	Real	Real
TANHF(X)	Hyperbolic tangent X radians	Real	Real
SQRTF(X)	Square root of X	Real	Real
LOGF(X)	Natural log of X	Real	Real
EXPF(X)	e to *Xth* power	Real	Real
SIGNF(X_1,X_2)	Sign of X_2 times $\|X_1\|$	Real	Real
XSIGNF(i_1,i_2)	Sign of i_2 times $\|i_1\|$	Integer	Integer
DIMF(X_1,X_2)	IF $X_1 > X_2$: $X_1 - X_2$; If $X_1 \leqq X_2$: 0	Real	Real
XDIMF(i_1,i_2)	If $i_1 > i_2$: $i_1 - i_2$; If $i_1 \leqq i_2$: 0	Integer	Integer

Form	*Definition*	*Actual Parameter Type*	*Mode of Result*
CUBERTF(X)	Cube root of X	Real	Real
FLOATF(I)	Integer to Real Conversion	Integer	Real
RANF(N)*	Generate Random Number	−Real, −Integer	Real
	(Repeated Executions give uniformly distributed numbers)	+Real, +Integer	Integer
XFIXF	Real to Integer Conversion	Real	Integer
POWER(X_1X_2)	$X_1^{X_2}$	Real, Real	Real
ITOJ(I,J)	I^J	Integer, Integer	Integer
XTOI(X,I)	X^I	Real, Integer	Real
ITOX(I,X)	I^X	Integer, Real	Real
LENGTHF(i)	Number of words read on unit i	Integer	Integer

*Any compiled calls to RANF(N), used as a function, treat the resulting value as real.

E. QUANTITIES AND WORD STRUCTURE

FORTRAN-66 manipulates floating point and integer quantities. Floating point quantities have an exponent and a fractional part. The following classes of numbers are floating point quantities.

REAL

Exponent and sign 11 bits; fraction and sign 49 bits; range of number, (in magnitude) $10^{-308} \leq N \leq 10^{308}$ and zero; precision approximately 14½ decimal digits.

DOUBLE

Each half is independently packed.

COMPLEX

Two reals as defined above.

Integer quantities do not have a fractional part. The following classes of numbers are integer quantities:

INTEGER

Represented by 60 bits, first bit is the sign; range of number (in magnitude) $0 \leq N \leq 2^{60}-1$; precision is up to 17½ decimal digits.

LOGICAL

1 bit represents the value TRUE.
0 bit represents the value FALSE.

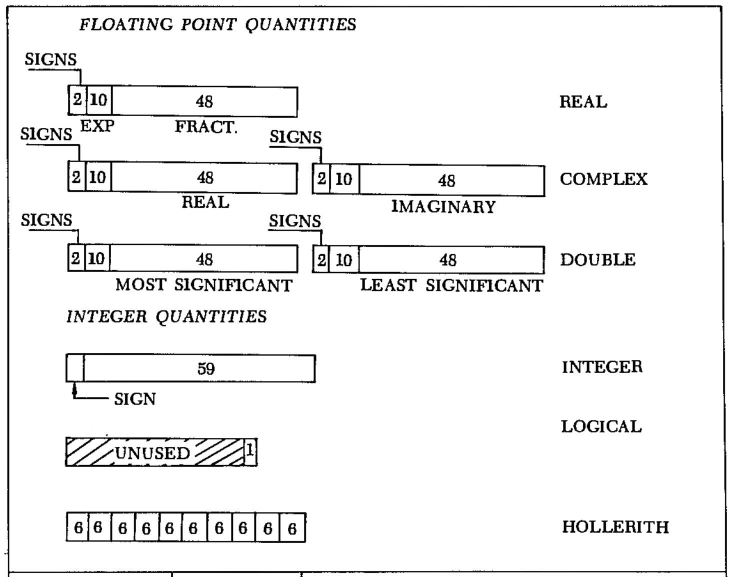
HOLLERITH

Binary coded decimal (BCD) representation treated as an integer number.

A FORTRAN-66 program may contain any or all of these classes of numbers in the forms of constants, variables, elements of arrays, evaluated functions and so forth. Variables, arrays and functions are associated with types assigned by the programmer. The type of a constant is determined by its form.

WORD STRUCTURE

The word structure of the quantities in FORTRAN-66 is shown below:



TYPE	λ*	REMARKS
COMPLEX	2 WORDS	CONSECUTIVE MEMORY LOCATIONS
DOUBLE	2 WORDS	CONSECUTIVE MEMORY LOCATIONS
REAL	1 WORD	
INTEGER	1 WORD	
HOLLERITH	1 WORD	6 BITS/CHARACTER
LOGICAL	1 BIT	

*ELEMENT LENGTH

F. PROGRAM SEGMENTATION

In general, the complex for a central processor program is assumed to be made up of control programs, subroutines, and common data blocks. The initiating control program, any common subroutines and data blocks comprise a permanent segment in core. Any subsequent control programs, subroutines and their data blocks are arranged dynamically in core in segments according to requests encountered during execution, and as defined by segmentation control cards.

The compiling process handles all programs and subroutines individually. The routines are compiled separately and, in binary form, are put together with segmentation specifications at execute time. Linkage betwen segments and between routines is handled at load time. Although a routine may appear in any number of segments, only one copy need be compiled and placed with the job.

Definition of Terms

Basic Segment — a fixed arrangement of a control program, subroutines, and common data blocks.

Normal Segment — an arrangement of a control program, subroutines, and common data read into central memory dynamically as required. It is defined by a segment control card and loaded by means of the LOAD statement. Normal segments may be overlayed.

Control Program — defined with a PROGRAM card and is the only executable program within a basic segment or a normal segment.

Subroutine — a routine defined by a SUBROUTINE card and executed by means of a CALL statement.

LOAD — is a macro defined as the overlay segment request.

The contents of a segment are provided by segmentation control cards at load time and are specified as a combination of control programs, subroutines, and other segments. The overlay operation when executed does not disturb the contents of the basic segment. However, it does destroy all other requested segments operated prior to the loading of the overlay. Overlay requests may occur without restriction in any segment and are coded in line where the decision is reached that an overlay is required.

Two independent segmentation concepts are provided under a single programming mechanism. The first method allows a basic segment, which resides permanently, in core, to initiate loading, and to control routing to the various subroutines in the job. Under this concept, control flows back and forth between the basic segment and any routine in the other segments, but always returns to the basic segment prior to the initiation of an overlay. When an overlay is initiated, control continues to the next instruction in line (no control transfer occurs).

The second method provides a program chaining operation. Under this method each successive overlay has its own control program and provides control routing through the various routines in that overlay. Each new overlay destroys the preceding one. Since loading is initiated by the overlay, it is necessary that a control transfer be made in conjunction with the load request.

An overlay request may occur in any segment. However, a transfer address must be provided if the request is made from other than the basic segment. The transfer address, if required, is taken to be that of the control program which has a trace of asterisks all the way through to the LOAD statement. An example of a segmented program and the tracing of control through the segments with asterisks follows:

EXAMPLE: (Control in Normal Segments)

Assume the basic segment is loaded consisting of non-executable statements defining common data and storage areas and a load request:

```
LOAD      * SEG1 *
```

SEG1 is defined as:

```
SEG1 = *PROGRAM A*, SUBROUTINE
          A1, SUBROUTINE A2
```

After the load is accomplished, control is shifted to Program A as a result of asterisks around SEG1 and Program A in the definition of SEG1.

Assume a load request is given in SEG1:

LOAD   * SEG2 *

SEG2 = *PROGRAM B*, SUBROUTINE B1, SUBROUTINE B2

The load is then performed and control turned over to Program B.

Suppose an overlay request exists within SEG2 which includes both segments SEG1 and SEG2, plus Program C. In addition, suppose control is to be shifted back to Program A after SEG3 has been loaded. The specification of the third segment may take one of several forms:

1. SEG3 = *PROGRAM A*, SUBROUTINE A1, SUBROUTINEA2, PROGRAM B, SUBROUTINE B1, SUBROUTINE B2, PROGRAM C
2. SEG3 = /*SEG1*/, PROGRAM B, SUBROUTINE B1, SUBROUTINE B2, PROGRAM C
3. SEG3 = /*SEG1*/, /SEG2/, PROGRAM C

where slashes indicate the enclosed is a segment name.

Upon a LOAD *SEG3* statement, each of these three forms will produce the same core arrangement and transfer control to Program A since it is the only routine in this segment specification which has an asterisk trace through the segment definitions to the LOAD statement. Program A is asterisked (definition 1 above contains asterisks around Program A, definitions 2 and 3 above have asterisks around SEG1, which contains Program A enclosed with asterisks), SEG3 is asterisked in the LOAD statement.

However, if control is desired for Program C, SEG3 may then be defined as:

SEG3 = /SEG1/, /SEG2/, *PROGRAM C*

or either of the other two forms may be used with Program C enclosed with the asterisks.

A transfer address is not necessary when the basic segment maintains control throughout the execution of the object program. In this instance, segments consist of subroutines which are executed by means of a CALL statement. An example of a segmented program with control residing in the basic segment follows:

EXAMPLE: (Control in Basic Segment)

Assume the basic segment is loaded and executing. At some point within the basic segment, a load request is coded:

LOAD   SEG4

SEG4 is defined as:

SEG4 = SUBROUTINE A, SUBROUTINE A1, SUBROUTINE A2

After the load is accomplished, all programming conventions relative to subroutine calls and communications may be followed. However, when the tasks of the routines in the normal segment are completed, control always reverts to the calling program in the basic segment which may then request the loading of another segment.

With reference to these examples, general characteristics of segment specifications are stated as follows:

1. Segments may be defined as a combination of routines and other segments. There is no limit on the depth used in the specification. Segment hierarchies may be conveniently defined as specifying large and highly overlapping segments.
2. There is no requirement that a one-to-one correspondence exist between segment definitions and segments referenced in LOAD statements. Extra segments may be defined which contain common groupings of routines and these segments may then be used to define the larger segments. Each segment referenced by a LOAD statement must be defined by a segmentation card; if not, the entire job is aborted.
3. With the chaining method, control is passed to a program in the loaded segment by an unbroken chain of asterisks from the LOAD statement through the segment specifications to the program name. This provides the necessary control routine capability.

4. There is no requirement that program or subroutine names appear uniquely in the combination of segments that define a segment. The union of the segments designates the routines to be loaded. This assures that there is no loss in core utilization due to name redundancies.

5. Segment identifiers used in the definition of other segments are surrounded by slash marks (/) to distinguish them from program and subroutine names. If a segment name is enclosed in asterisks, the asterisks are placed inside the slash marks.

The following restrictions are made:

1. If the LOAD statement specifies a transfer of control, one and only one program in the loaded segment must be designated as recipient of the control by meeting characteristic number 3.

2. No mixing of the two methods is permissible although a one-time switch may be made from the basic segment concept to the chaining concept. Once the chaining process is used, there is no way to return control to the PROGRAM in the basic segment.

A core map taken at the completion of the loading of the basic segment would show the control programs, the programmer subroutines in the basic segment, one copy of common data blocks, and one copy of library subroutines and functions referenced by these basic segment routines. The block of memory required for this segment is permanently reserved. The next location after this block is the initial address for all normal segments called by a LOAD statement.

If there are additional segments specified to accompany the basic segment in the initial load or when an overlay request occurs, these segments are brought in from the disk and relocated at the initial overlay address. Addresses for common blocks referenced in the segments are determined from their location within the basic segment, or they are assigned and inserted where appropriate. Also one copy of any library subroutine, not previously required or loaded, is added to the program and linkages for all routines are then made.

When a transfer is indicated by the segment specification (asterisks around NAME), a similar name form is sought from the segment name list and control is routed to its single entry point.

G. FORTRAN ERROR PRINTOUTS

Listed below are the diagnostie error printouts eurrently available with FORTRAN-66. A one (1) before the statement indieates that no further compilation takes plaee. However, the subsequent FORTRAN statements are diagnosed for errors. A zero (0) before the statement indicates that the program will be eompiled, but execution is likely to be subject to error.

```
1*********UNRECOGNIZABLE STATEMENT
1*FUNCTION STATEMENT FORMAT ERROR
1*SUBROUTINE STATEMENT FORMAT ERROR
1***SUBROUTINE OR VARIABLE NAME TOO LONG
1*MORE THAN 20 ENTRIES
1**ERROR IN CALL STATEMENT****
1**ILLEGAL CHARACTER
1**MORE THAN 63 PARAMETERS IN LIST****
1**DUPLICATION IN PARAMETER LIST***
0**NO HEADER CARD FOUND ASSUME FORTRAN MODE**
1**MORE THAN ONE HEADER CARD**
1**DECLARATIVE STATEMENT OUT OF ORDER***
0**NON-EXECUTABLE PROGRAM*****
1*MORE THAN 9 CONTINUATIONS
1 RETURN STATEMENT USED IN PROGRAM MODE
1*******UNCLOSED DO LOOP DO [N] I = ETC.******
1**DUPLICATE ENTRY NAME**
1*DIMENSION NOT FOUND, VARIABLE********
1*DIMENSION EXCEEDED, VARIABLE********
1*SUBSCRIPT NOT FOUND, VARIABLE********
1******ARITHMETIC STATEMENT FORMAT ERROR
1*NO EQUAL SIGN FOUND IN ARITHMETIC STATEMENT
1*LEFT SIDE ERROR
1*ELEMENT NOT AN ARRAY
1*FUNCTION NOT ON LEFT
1****ILLEGAL ALPHA-CHARACTER APPEARING IN NUMBER FIELD
1*ILLEGAL PERIOD (.) OCCURRING
1*******UNBALANCED PARENTHESIS
1****ILLEGAL CHARACTER IN OCTAL NUMBER FIELD
1***ERROR IN GO TO STATEMENT**
1*ASSIGN STATEMENT FORMAT ERROR
1**IF STATEMENT FORMAT ERROR
1*ERROR IN SENSE LIGHT STATEMENT*
1****ERROR IN DO STATEMENT
1***INDEX USED BY ANOTHER DO
1*DO STATEMENT ENDED BY ********, ILLEGALLY NESTED***
1**DO STATEMENT NEXT CONTAINS MORE THAN 50*****
0*DO STATEMENT ENDED BY A TRANSFER STATEMENT
1**ERROR IN I/O STATEMENT*****
1ARRAY XXXX IS NOT A SUBROUTINE FORMAL PARAMETER
1SUBSCRIPT XXXX IS NOT A SUBROUTINE FORMAL PARAMETER
1COMMON/EQUIVALENCE CONFLICT, VARIABLES ... XXXX AND ... YYYY
1LITERAL TABLE OVERFLOW
1OCTAL TO DISPLAY CONVERSION ERROR
```

```
1EQUIVALENCE STATEMENT REFERENCE CONFLICT, VARIABLE XXXX REFERENCES
  VARIABLE YYYY IN MORE THAN ONE WAY. FIRST REFERENCE PREVAILS
1EQUIVALENCE STATEMENT REFERENCE CONFLICT, VARIABLE XXXX REFERENCES ITSELF
1FORMAL PARAMETER XXXX APPEARS IN A COMMON STATEMENT
0OBSERVE EQUIVALENCE STATEMENT SUBSCRIPT RULES
1FORMAT ERROR
0COMMA AT END OF STATEMENT
1TYPE STATEMENT CONFLICT, VARIABLE XXXX
0DUPLICATE TYPE STAEMENT, VARIABLE XXXX
1INCOMPLETE STATEMENT
1TOO MANY SUBSCRIPTS, VARIABLE XXXX
1COMMON STATEMENT CONFLICT, VARIABLE XXXX
1PRIOR SUBSCRIPTS WILL BE USED
1DIMENSION STATEMENT CONFLICT, VARIABLE XXXX
1COMMON STATEMENT/DIMENSION STATEMENT CONFLICT, VARIABLE XXXX
0DIMENSION STATEMENT SUBSCRIPTS WILL BE USED
0DUPLICATE SUBSCRIPT SET ASSIGNED TO VARIABLE XXXX
1SUBSCRIPT, VARIABLE XXXX IS TOO BIG
1SUBSCRIPT CONFLICT OR DUPLICATE, VARIABLE XXXX
1FORMAL PARAMETER XXXX APPEARS IN AN EQUIVALENCE STATEMENT
0A MINIMUM OF TWO VARIABLES MUST OCCUR IN EACH EQUIVALENCE GROUP
1FUNCTION CALLING SEQUENCE ERROR.FUNCTION          XXXXX
1LEFT SIDE NOT TYPE LOGICAL, WITH RIGHT SIDE RELATIONALS.
1FAILURE OF εQ B6 B7 K , TO JUMP ON EQUALITY.
1STATEMENT FUNCTION TABLE FULL.
1NUMBER OF PARAMETERS DOES NOT AGREE WITH STATEMENT FUNCTION DEFINITION.
1UNIARY SIGN FORMAT ERROR.
1RIGHT SIDE RESULTS MASSING. $$$$$$$$$$$$$$$$$$$$$$$$
1MULTIPLE RIGHT SIDE RESULTS. $$$$$$$$$$$$$$$$$$$$$$$
1UNKNOWN ELEMENT AS AN OPERAND = $$$$$$$$$$$$$$   XXXXX
1ARITHMETIC OPERANDS ARE NOT SAME TYPE. $$$$$$$
1OPERANDS HAVE ILLEGAL TYPE 6. $$$$$$$$$$$$$$$$$$$$$$
1OPERANDS HAVE ILLEGAL TYPE 7. $$$$$$$$$$$$$$$$$$$$$$
1ILLEGAL TYPE 6, FOR CONVERSION. $$$$$$$$$$$$$$$$$$$$
1ILLEGAL TYPE 7, FOR CONVERSION. $$$$$$$$$$$$$$$$$$$$$
1ILLEGAL TYPE 6, FOR CONVERSION. TO OCTAL. $$$$$$$$$
1ILLEGAL TYPE 7, FOR CONVERSION. TO OCTAL. $$$$$$$$$
1ILLEGAL TYPE 6, FOR CONVERSION. TO LOGICAL.  $$$$$$
1ILLEGAL TYPE 7, FOR CONVERSION. TO LOGICAL.  $$$$$$
1ILLEGAL TYPE 6, FOR CONVERSION. TO INTEGER.  $$$$$$
1ILLEGAL TYPE 7, FOR CONVERSION. TO INTEGER.  $$$$$$
1ILLEGAL TYPE 6, FOR CONVERSION. TO REAL. $$$$$$$$$$$
1ILLEGAL TYPE 7, FOR CONVERSION. TO REAL. $$$$$$$$$$$
1ILLEGAL TYPE 6, FOR CONVERSION. TO DOUBLE.$$$$$$$
1ILLEGAL TYPE 7, FOR CONVERSION. TO DOUBLE.$$$$$$$
1ILLEGAL TYPE 6, FOR CONVERSION. TO COMPLEX.  $$$$$
1ILLEGAL TYPE 7, FOR CONVERSION. TO COMPLEX.  $$$$$
1DUPLICATE STATEMENT FUNCTION =      XXXXX
0ZERO SUBSCRIPT CHANGED TO ONE FOR VARIABLE     XXXXX
```

CONTROL DATA SALES OFFICES

ALAMOGORDO • ALBUQUERQUE • ATLANTA • BILLINGS • BOSTON • CAPE CANAVERAL • CHICAGO • CINCINNATI • CLEVELAND • COLORADO SPRINGS DALLAS • DAYTON • DENVER • DETROIT • DOWNEY, CALIFORNIA • HONOLULU HOUSTON • HUNTSVILLE • ITHACA • KANSAS CITY, KANSAS • LOS ANGELES MADISON, WISCONSIN • MINNEAPOLIS • NEWARK • NEW ORLEANS • NEW YORK CITY • OAKLAND • OMAHA • PALO ALTO • PHILADELPHIA • PHOENIX PITTSBURGH • SACRAMENTO • SALT LAKE CITY • SAN BERNARDINO • SAN DIEGO • SEATTLE • ST. LOUIS • WASHINGTON, D.C.

ATHENS • CANBERRA • DUSSELDORF • FRANKFURT • THE HAGUE • HAMBURG JOHANNESBURG • LONDON • MELBOURNE • MEXICO CITY (REGAL ELECTRONICA DE MEXICO, S.A.) • MILAN • MONTREAL • MUNICH • OSLO • OTTAWA PARIS • SAVYON • STOCKHOLM • STUTTGART • SYDNEY • TOKYO (C. ITOH ELECTRONIC COMPUTING SERVICE CO., LTD.) • TORONTO • ZURICH

8100 34th AVE. SO., MINNEAPOLIS, MINN. 55440

CONTROL DATA CORPORATION

Pub. No. 60101500B

Litho in U.S.A.